Efficient 2D SLAM for a Mobile Robot with a Downwards Facing Camera

Christian Colliander
christian.colliander@gmail.com

Supervisor
Magnus Oskarsson
magnuso@maths.lth.se

Examiner
Anders Heyden
heyden@maths.lth.se

Abstract

As digital cameras become cheaper and better, computers more powerful, and robots more abundant the merging of these three techniques also becomes more common and capable. The combination of these techniques is often inspired by the human visual system and often strives to give machines the same capabilities that humans already have, such as object identification, navigation, limb coordination, and event detection. One such field that is particularly popular is that of SLAM, or Simultaneous Localization and Mapping, which has high-profile applications in self-driving cars and delivery drones.

This thesis proposes and describes an online SLAM algorithm for a specific scenario: that of a robot with a downwards facing camera exploring a flat surface (e.g., a floor). The method is based on building homographies from robot odometry data, which are then used to rectify the images so that the tilt of the camera with regards to the floor is eliminated, thereby moving the problem from 3D to 2D. The 2D pose of the robot in the plane is estimated using registrations of SURF features, and then a bundle adjustment algorithm is used to consolidate the most recent measurements with the older ones in order to optimize the map.

The algorithm is implemented and tested with an AR.Drone 2.0 quadcopter. The results are mixed, but hardware seems to be the limiting factor: the algorithm performs well and runs at $5-20$ Hz on a i5 desktop computer; but the bad quality, high compression and low resolution of the drone's bottom camera makes the algorithm unstable and this cannot be overcome, even with several tiers of outlier filtering.

Acknowledgement

I would like to thank my supervisor, Magnus Oskarsson, for valuable input, ideas, and help. I would also like to thank Anders Robertson for helping me with the quadcopter and with getting set up in the Robotics Lab.

Contents

Chapter 1

Introduction

Computer Vision is an interdisciplinary field concerned with automatically extracting high-level information about an object or scene from one or more images. Common usage scenarios include scene reconstruction, object recognition, event detection, and camera pose estimation. In many ways, computer vision aims to give computers capabilities inspired by the human vision system, but with the ability to operate with super-human speed, precision, and acuteness. The field is closely tied to machine learning and sometimes the line between the two disciplines becomes blurred, as in the case of convolutional neural networks. The two disciplines are often employed in tandem, where the high-level information extracted via computer vision is fed into some sort of decision making system. Currently, computer vision is seeing an explosion in terms of research, investment and applications, largely ushered in by the recent advances in computing power, digital cameras, robotics and machine learning. Some of the more high-profile uses are self-driving cars, autonomous robots and large-scale data classification and annotation.

Quadcopters, or *drones* in the common parlance, are also seeing an explosion in terms of their commodity and application. Some of of the driving factors here are the same as with computer vision[1]: faster and cheaper computing power; better and smaller digital cameras; but advances in batteries and MEMS (MicroElectroMechanical Systems) in recent years are also a driving factor. Quadcopters come in a wide range of sizes and with a wide range of intended uses, from small, thumb-sized drones intended as fun gadgets [1],

[1] Also, there is significant synergy between the fields of computer vision, machine learning and robotics.

to drones large enough to - and intended to - fit a human [28]. Another use for quadcopters is mapping; here the agility and mobility of quadcopters can be levied to great effect for quickly exploring a scene, even from angles that are inaccessible to humans or other types of robots.

SLAM (Simultaneous Localization And Mapping) is the problem within computer vision of simultaneously constructing and updating a map of a previously unknown environment while also keeping track of an agent's location within that environment. This is often done in a back-and-forth manner [13], where new measurements are first fitted to the existing map, and then the map is updated (using the measurements as constraints) to obtain a map that maximizes the likelihood of the measurements. On a slightly lower level, SLAM can be broken down into coarser parts: *landmark extraction*, *data association*, *state estimation*, and *state and map update* [37]. In the first part, new data from the agent's sensors are processed to extract high-level, invariant (more or less) landmarks. In the second part, associations are made between the newly extracted landmarks and previously detected landmarks. In the third part, a model is fitted to these associations (and optionally other data as well, like odometry) in order to get some estimate of the agent's state. In the fourth and final part, poses and landmarks are retroactively updated, taking into account the latest set of measurements.

There is a wide variety of measurement devices/sensors that can be used for SLAM, for example: laser rangefinders [13], Lidar [18], thermal cameras [18], sonar [7], radar [19], regular cameras (either monocular [6, 9, 8, 14] or stereo [35, 15]), and even tactile "whiskers" [10]. SLAM using regular cameras ("Visual SLAM") is a common method, largely due to the recent advances in digital camera techniques and their resulting broad adaptation, mentioned earlier in this section.

This is where this thesis comes in: the original aim of the thesis was to develop a Visual SLAM-algorithm for a particular scenario and a relatively inexpensive drone. In the scenario the drone maps a planar surface, while moving in a parallel plane, using a single camera directed towards the first plane. A real world scenario would be a drone navigating and exploring a building by looking at the floor. The scenario is simplified from the full 3D mapping problem (where both the landmarks and the agent moves in 3D space) and so it was hoped that the algorithm could be made quite effective,

performance-wise.

Partway into the project it was discovered that the hardware (most notably the downwards facing camera of the selected drone, an AR.Drone 2.0) was going to be a limitation. At that point the goal was redefined so that the aim was to make the SLAM-algorithm as good as possible, given the hardware limitations.

1.1 Related work

This thesis was based on earlier work by Wadenbäck and Heyden [41] [40], Brange [4], and Rudbeck [38].

Wadenbäck and Heyden developed [41] - and then refined [40] - a method for recovering the tilt and motion (translation + in-plane rotation) of a camera using inter-image homographies, under the assumption of planar motion and constant tilt. These homographies could for example come from point correspondences between the images and some robust fitting algorithm, e.g., a RANSAC[2]-based solver for the 3×3 homography $\mathbf{H}$.

The works by Brange and Rudbeck were similar to each other in that both dealt with a camera moving in one plane and taking images of another, parallel plane, with the camera having some constant tilt that was compensated for using a rectification homography. Both authors also recovered the pose/-planar motion of the camera using an SVD[3]-based least-squares point-set registration algorithm from Arun, Huang, and Blostein [2]. Brange [4] then used these poses and images, along with loop closure detection and bundle adjustment to build an offline 2D map construction tool. Rudbeck [38] used the poses along with a pre-built map of the environment (using Brange's algorithm from [4]) to build a navigation algorithm for a wheeled robot equipped with a downwards facing, tilted camera.

In both Brange's and Rudbeck's work the rectification homography was estimated once and then used for all images taken by the camera. Both authors used the algorithm developed by Wadenbäck and Heyden [40] to estimate

[2]Random Sample Consensus.
[3]Singular-Value Decomposition

the tilt of the camera and then construct a homography that compensated for that tilt. In the scenario studied in this thesis the tilt of the camera was not constant, as a quadcopter with a rigidly mounted camera was used. The drone banked, pitched, and rolled as it flew around, and thus a new homography had to be calculated for each image using odometry readings from the quadcopter.

The bundle adjustment used in [4] was heavily inspired by an algorithm developed by Konolige et al. [13] for sparse bundle adjustment of 2D poses. That algorithm also inspired the bundle adjustment in this thesis, but significant changes were made to the algorithm in order to move it away from pose-pose based constraints and instead use pose-feature based constraints. In [13] constraints were established between poses and then the poses were shifted around together with all their associated features as the map was optimized. In this thesis the bundle adjustment was a bit closer to "normal" bundle adjustment (where 3D points and camera poses are simultaneously optimized in order to maximize the likelihood of the measurements from the 2D images): the optimization was done over a set of point-feature correspondences in order to simultaneously optimize the location of the features in the map and the poses of the drone.

Another significant difference from the previous work by Brange and Rudbeck was that the map building there was done *offline*: there, a sequence of pre-collected images were pairwise registered to each other and then loop closure detection and bundle adjustment was used to optimize the configuration of the images using more long-term correspondences. The final image registrations were then used to create one large "map-image" from which the final features that then made up the actual map were extracted. In this project the map building was done *online*, using live data from a drone navigating a scene. The goal was to simultaneously locate the drone in the scene and build a map of the scene, i.e., the resulting algorithm was one of Simultaneous Localization And Mapping (SLAM) and the map needed to be available and accurate throughout the entire run. In this scenario performance was very important: the execution time of each iteration couldn't be too long, otherwise the map could become to outdated between iterations.

Chapter 2

Theory

2.1 Computer Vision

2.1.1 The pinhole camera model

The pinhole camera model is the most commonly used mathematical model of a camera. The name of the model comes from the type of camera it models; a pinhole camera has the shape of a box and light from the scene to be photographed enters the camera through a small hole - the pinhole - at the front of the camera and produces an image at the back wall of the camera (see Fig. 2.1).

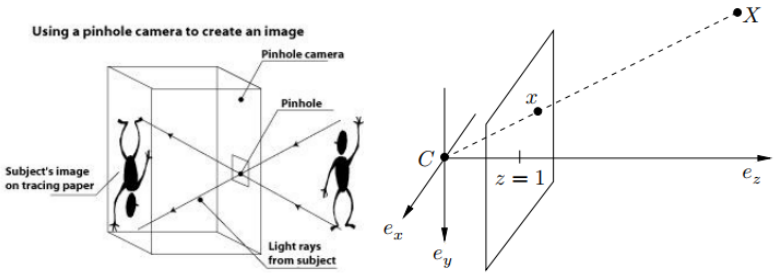


Figure 2.1: A pinhole camera (left), and the mathematical model inspired by the pinhole camera (right). The image is taken from [31].

To create the mathematical model of the pinhole camera a right-hand Cartesian coordinate system $\{\mathbf{e}_x, \mathbf{e}_y, \mathbf{e}_z\}$ is introduced, with the origin $\mathbf{C} = (0, 0, 0)$

- representing the camera center - located at the pinhole (the directions of the axes are shown in Figure 2.1). This coordinate system is referred to as the *camera coordinate system*. A projection $\mathbf{x} = (x_1, x_2, 1)$ of a scene point $\mathbf{X} = (X_1, X_2, X_3)$ is then generated by forming a line between $\mathbf{X}$ and $\mathbf{C}$ and finding the intersection with the plane $z = 1$ (the so-called *image plane*). The line $\mathbf{e}_z$, perpendicular to the image plane and originating from $\mathbf{C}$, is the *optical axis*, or viewing direction of the camera. The fact that the image plane is placed in front of the camera center (as opposed to behind, like in a real pinhole camera) has the effect that the image won't appear upside down, which simplifies things a bit.

The *viewing ray*, $\mathbf{X} - \mathbf{C}$, can be parameterized via

$$\mathbf{C} + s(\mathbf{X} - \mathbf{C}) = s\mathbf{X}, \quad s \in \mathbb{R} \tag{2.1}$$

$s\mathbf{X}$ is then a vector pointing from $\mathbf{C}$ to $\mathbf{X}$ with length given by the product of s and $|\mathbf{X}|$. Then an s is chosen s.t. the resulting projection, $\mathbf{x}$, of $\mathbf{X}$ onto the image plane becomes:

$$s\mathbf{X} = \mathbf{x} = s \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \frac{1}{X_3} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} X_1/X_3 \\ X_2/X_3 \\ 1 \end{bmatrix} \tag{2.2}$$

In other words, the projection onto the image plane is found[1] by element-wise dividing the scene point's coordinates with the third scene point coordinate[2] in the camera coordinate system, i.e., the *depth*. This operation is sometimes called *z-dividing*.

The image plane is embedded in $\mathbb{R}^3$, with the center of the image located in $(0, 0, 1)$ and lengths given in whatever unit the camera coordinate system uses (e.g., meters). This contrasts with images from real cameras, which typically have coordinates measured in pixels and $(0, 0)$ located in the upper left corner ("row 0, column 0"). This coordinate system is known as the *image coordinate system*.

[1] Assuming that the z-coordinate isn't 0.

[2] I.e., multiplying with $s = \frac{1}{X_3}$.

To move between the camera coordinate system and the image coordinate system a mapping is introduced, represented by an invertible, upper-triangular 3×3 matrix $\mathbf{K}$, called the *intrinsic matrix*, see (2.3). The matrix contains the so-called *intrinsic parameters* of the camera: the *focal length*, f, which re-scales image coordinates into pixels; the *principal point*, (x_0, y_0), which translates image coordinates so that $(0, 0)$ is in the upper left corner; the *aspect ratio*, γ, which controls how coordinates are scaled differently in the x- and the y-direction; and the *skew*, s, which corrects for (the rare case of) tilted pixels.

$$\mathbf{K} = \begin{bmatrix} \gamma f & sf & x_0 \\ 0 & f & y_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

The skew is zero for most cameras, and for such normal, zero-skew cameras it is common to simplify (2.3) and replace γf with f_x and f with f_y. An example is shown in (2.4) of a point $\mathbf{X}$ in the camera coordinate system being transformed to pixel coordinates in the image coordinate system, using a zero-skew camera (i.e., $s = 0$).

$$\begin{aligned} \mathbf{X} &= \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = X_3 \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = X_3 \mathbf{x} \propto \mathbf{x} \\ \mathbf{Kx} &= \begin{bmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = \begin{bmatrix} f_x x_1 + x_0 \\ f_y x_2 + y_0 \\ 1 \end{bmatrix} \end{aligned} \tag{2.4}$$

Moving the other way, i.e., from pixel coordinates in the image coordinate system to projected coordinates $\mathbf{x}$ in the camera coordinate system's image plane, is done by applying the inverse of the intrinsic matrix, K^{-1}, to the pixel coordinates. Restoring scene coordinates $\mathbf{X}$ from projected coordinates $\mathbf{x}$ is more complex though, and is outside the scope of this paper[3].

The camera coordinate system is often enough when only dealing with a single image of a given scene, but when there are multiple images of the same scene from multiple viewpoints (e.g., pictures taken from a camera moving around in the scene) there needs to be a way to model camera movements. To this end a *world-, or global coordinate system*, $\{\mathbf{e}_x^w, \mathbf{e}_y^w, \mathbf{e}_z^w\}$, is introduced.

[3]See [11] for information about how this can be done.

In this coordinate system a camera can undergo translation and rotation, i.e., *rigid transformations*. The translation is specified using a vector $\mathbf{t} \in \mathbb{R}^3$ and the rotation using a 3×3 rotation matrix $\mathbf{R}$, which fulfills $\mathbf{R}^T\mathbf{R} = \mathbf{I}$ and $det(\mathbf{R}) = 1$. The relation between a point's coordinates in the global coordinate system, $\mathbf{X}^w$, and its coordinates in the camera coordinate system, $\mathbf{X}$ is given by (2.5).

$$\mathbf{X} = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \mathbf{R}\mathbf{X}^w + \mathbf{t} = \mathbf{R} \begin{bmatrix} X_1^w \\ X_2^w \\ X_3^w \end{bmatrix} + \mathbf{t} \tag{2.5}$$

This can be simplified into a single matrix operation if the world coordinate $\mathbf{X}^w$ is padded with a 1 as a fourth coordinate:

$$X_3 \begin{bmatrix} X_1/X_3 \\ X_2/X_3 \\ 1 \end{bmatrix} = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \begin{bmatrix} X_1^w \\ X_2^w \\ X_3^w \\ 1 \end{bmatrix} \tag{2.6}$$

Combining the transformation from the global coordinate space to the camera coordinate space with the mapping to the image coordinate space and the projection to the image plane results in the *camera equations*:

$$\lambda \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix}}_{\mathbf{P}} \underbrace{\begin{bmatrix} X_1^w \\ X_2^w \\ X_3^w \\ 1 \end{bmatrix}}_{\mathbf{X}^w} \tag{2.7}$$

where the 3×4 matrix $\mathbf{P}$ is called the *camera matrix* and λ is the depth of the imaged point. The camera equations can be written more concisely as 2.8.

$$\lambda\mathbf{x} = \mathbf{P}\mathbf{X}^w \tag{2.8}$$

2.1.2 Homographies

Projective transformations, or *homographies*, are invertible mappings $\mathbf{H}$, $\mathbb{P}^n \mapsto \mathbb{P}^n$, relating one set of points with another set of points, i.e.,

$$\lambda \mathbf{x} = \mathbf{H}\mathbf{y} \tag{2.9}$$

where $\mathbf{x} \in \mathbb{R}^{n+1}$ and $\mathbf{y} \in \mathbb{R}^{n+1}$ are homogeneous coordinates of points in $\mathbb{P}^n$, $\mathbf{H}$ is an invertible $(n+1) \times (n+1)$ matrix, and λ is a scale factor (i.e., the homography is unique up to scale). Because of this scale ambiguity it is necessary to do a z-divide again if a homography is applied to image coordinates and the output is intended to be in image coordinates too.

A notable use of homographies is to transform points from one plane to another plane - like in this thesis, where a homography maps points from a tilted plane/image to a plane that is perpendicular to the camera's optical axis.

2.1.3 Lens Distortion

The relation between world coordinates and image coordinates does not account for the lens of real, physical cameras. For most real cameras the lens distorts the image to such a degree that it negatively impacts the results of computer vision applications if it is not corrected for.

If the distortion of a camera is known (which it will be after the camera has been *calibrated*, see Section 3.3) the distortion can be removed from its images. Commonly, two types of lens distortion are corrected for, which arise from the shape of the lens and its position with respect to the camera sensor [23]: *radial distortion*, which occurs when light bends more near the edges of the lens than near its center and *tangential distortion*, which occurs when the lens and the camera sensor are not parallel.

Radial distortion of image points is modeled and corrected for using the relations

$$\begin{aligned} x_{dist} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{dist} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{2.10}$$

where (x_{dist}, y_{dist}) are the distorted coordinates of an image point; (x, y) are the distortion-less coordinates of the image point; k_1, k_2 and k_3 are the radial distortion coefficients of the lens; and $r = \sqrt{x^2 + y^2}$ is the distance from the

image center of the point. For many lenses it is enough to only include 2 radial distortion parameters, the third parameter is often only required when dealing with extreme radial distortion, e.g., from wide-angle lenses.

Tangential distortion is modeled and corrected for using the relations

$$\begin{aligned} x_{dist} &= x + (2p_1xy + p_2(r^2 + 2x^2)) \\ y_{dist} &= y + (p_1(r^2 + 2y^2) + 2p_2xy) \end{aligned} \tag{2.11}$$

where (x_{dist}, y_{dist}) are the distorted coordinates of an image point; (x, y) are the distortion-less coordinates of the image point; p_1 and p_2 are the tangential distortion coefficients of the lens; and $r = \sqrt{x^2 + y^2}$ is the distance from the image center of the point.

2.1.4 SURF Features

Computer vision applications commonly use feature detectors and descriptors to find, describe and match points of interest in images. What a "point of interest" in an image is depends on the feature detector used, but some common types of features are corners and/or areas of high contrast.

An important aspect of feature detectors and descriptors is their ability to detect the same feature from different images. An ideal feature system is invariant to scale, rotation, illumination and transformations, so that the same point can be reliably re-identified across different images.

In this project the *Speeded-Up Robust Features* (SURF) scale- and rotation-invariant feature detector and descriptor, by Bay et al. [3], is used. The reason SURF features are used is because of how successful they were when used for very similar applications by Brange [4] and Rudbeck [38]. Brange even did a study of different types of feature detectors and descriptors[4] for a very similar scenario and floor texture where SURF and FAST[5] came out on top. After running both the SURF feature detector and the feature descriptor extractor

[4]Specifically, SIFT, SURF, FAST, BRISK, the Harris Corner Detector, and the Minimum Eigenvalue method were evaluated.

[5]SURF was found to be slightly faster than FAST, and FAST was found to be slightly more accurate than SURF.

the resulting features are pairs of length two coordinates in the image, (x, y), and a feature vector/descriptor (commonly of length 64).

2.1.5 RANSAC

Random Sample Consensus (RANSAC) is a method for robustly fitting models to a large dataset that contains many outliers/bad data points/false data points. The core idea is that if a small subset of the points are sampled at random enough times the probability that one of those subsets will be outlier-free becomes very large.

The outline of the algorithm is as follows:

1. Randomly select the minimum number of points from the dataset needed to fit the model and solve the problem using only those points.
2. Evaluate the error residuals for all points in the set using that solution and count the number of inliers. Inliers are points with error residuals smaller than some threshold.
3. Repeat the above two points a fixed number of times and select the solution/model that resulted in the largest amount of inliers.

The required number of iterations of the RANSAC algorithm in order to have a specific probability of getting an outlier free set can be calculated if the number of outliers in the set is known. In practice, more iterations are better (to a point) due to noise in most real data sets - which makes good model fitting dependent on more than just getting an outlier free set.

2.1.6 Levenberg-Marquardt

The Levenberg-Marquardt algorithm is an iterative method for solving non-linear least-squares problems. The method adaptively combines the *gradient descent*-method and the *Gauss-Newton*-method to improve stability and performance over using either method alone.

All three of these methods work similarly in that they are iterative, and in each iteration, given a residual $e(\mathbf{v})$ that should be minimized, an update to the variable vector $\mathbf{v}_{i+1} = \mathbf{v}_i + \Delta\mathbf{v}$ is computed. Ideally the update results in a smaller value of the objective function and the methods iterate

either until a local minimum is found or until some other criterion is fulfilled. The main difference between the methods is how the update $\Delta\mathbf{v}$ is computed.

First, some definitions: given a least squares problem of the form

$$\min_{\mathbf{v}} \; e(\mathbf{v}) = \min_{\mathbf{v}} \sum_{j=1}^{m} \|\mathbf{e}_i(\mathbf{v})\|^2 \tag{2.12}$$

where the error residuals $\mathbf{e}_i$ are the residuals from fitting some non-linear model to m observed data points. The Jakobian $\mathbf{J}$ of $\mathbf{e}$ w.r.t $\mathbf{v}$ at iteration i is then defined as

$$\mathbf{J} \equiv \mathbf{J}(\mathbf{v}_i) = \left.\frac{\partial \mathbf{e}}{\partial \mathbf{v}}\right|_{\mathbf{v}=\mathbf{v}_i} \tag{2.13}$$

where $\mathbf{e}$ is a vector containing all the error residuals $\mathbf{e}_i$. Using this notation the *gradient descent update* at iteration i is given by

$$\Delta\mathbf{v} = -\beta \mathbf{J}^T \mathbf{e} \tag{2.14}$$

Where β is some small scalar chosen so that $e(\mathbf{v}_{i+1}) < e(\mathbf{v}_i)$. Similarly, the *Gauss-Newton update* at iteration i is given by

$$\Delta\mathbf{v} = -\left(\mathbf{J}^T\mathbf{J}\right)^{-1}\mathbf{J}^T\mathbf{e} \tag{2.15}$$

and the *Levenberg-Marquardt update* at iteration i is given by

$$\Delta\mathbf{v} = -\left(\mathbf{J}^T\mathbf{J} + \lambda\mathbf{I}\right)^{-1}\mathbf{J}^T\mathbf{e} \tag{2.16}$$

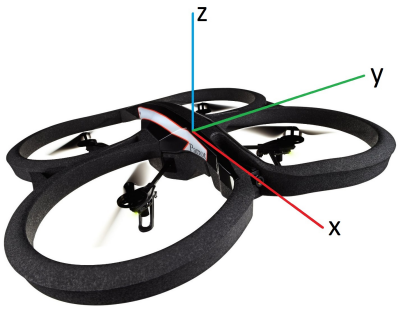
where λ is a variable scalar that is used to interpolate between more gradient descent-like behavior and more Gauss-Newton-like behavior. A large value for λ brings (2.16) closer to (2.14) (with $\beta = 1/\lambda$) and a small λ brings (2.16) closer to (2.15). It is common to start with a large λ and gradually reducing it when approaching the minimum value in order to take advantage of the rapid convergence of Gauss-Newton close to the minimum (and avoid its risk of instability when far away from the minimum) [32].

The Levenberg-Marquardt iterations can continue either until a local minimum is found, or until some other criteria is satisfied, e.g., that the updates start having diminishing returns w.r.t. decreasing the objective function, or that a certain number of iterations have been run.

2.2 The AR.Drone 2.0

The robot used in this project is a quadcopter called the AR.Drone 2.0 [34], manufactured by Parrot SA and launched in 2012. A photo of the drone can be seen in Figure 2.2a. The drone features an impressive suite of sensors and odometry: 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer, temperature and pressure sensors, ultrasound altimeter, and a front and a bottom camera [36]. The sensors that are of relevance to this project are the bottom camera, the ultrasound altimeter, and the gyroscope. Communication with the drone happens via WiFi.

(a) A photo of the AR.Drone 2.0 used for the experiments, resting on the floor in the LTH Robotics Lab.



(b) The local coordinate system of the drone. Roll is rotation around the x-axis, pitch is rotation around the y-axis, and yaw is rotation around the z-axis. For image credit, see [39].

Figure 2.2: The drone used in this project and its local coordinate system.

The bottom camera is a QVGA 60 fps camera with three color channels (RGB) and a resolution of 320×240, which is natively upscaled to 640×360. The camera is mainly intended to be used as a (native) ground speed sensor (using an optical flow algorithm) for automatic hovering and trimming. Because of this the camera is designed to focus at infinity and has no way of changing the focus.

The ultrasound altimeter measures the altitude at low altitudes (< 6 meters), and the air pressure sensor measures the altitude at higher altitudes, where the ultrasound altimeter can no longer measure the altitude. The ultrasound altimeter is more accurate than the air pressure altimeter. The altitudes

explored in this project (< 1 meter) are well within the ultrasound range.

The gyroscope is part of a 9-DOF MEMS (MicroElectroMechanical System) IMU (Inertial Measurement Unit). The IMU combines an accelerometer (3-DOF), a gyroscope (3-DOF), and a magnetometer (3-DOF). From the documentation [36] it is unclear how the readings from these three components (plus the ground speed measurements from the bottom camera + altimeter) are combined in order to estimate the drone's orientation, acceleration and velocity. It is also unclear whether the the drone has some built in safeguards against sensor drift or not.

The drone manoeuvres by independently varying the rotation speeds of the four rotors, which in turn causes the drone to throttle and/or change roll, pitch, and yaw. Unlike a helicopter, the pitch of the blades cannot be changed - all maneuvering is done by individually varying the thrust of each rotor. Figure 2.2b shows the local drone coordinate system and how the roll, pitch, and yaw angles relate to it.

2.2.1 Communicating with the drone

The communication with the drone is handled via ROS (Robot Operating System), a popular framework for programming and communicating with robots, which is elaborated on in the next section. The AR.Drone 2.0 hosts its own WiFi network over which all communication with the drone is conducted. A ROS driver (written in C++) is offered via the `ardrone_autonomy` package [25]. The ROS driver is based on the official ARDrone SDK 2.0 [36], which in turn is written mostly in C. After connecting to the drone's WiFi network a ROS driver node, `ardrone_driver`, from the above mentioned package needs to be launched. The node then abstracts the communication with the drone using ROS topics, services and a parameter server. Because of this abstraction the communication with the node and the communication with the drone are different things and they have different rates and timings, which are controlled via the driver. These update frequencies are set when the `ardrone_driver` node is launched and for this project the default values were used: the drone transmits data at 200 Hz, and the driver caches data and sends it at 50 Hz.

When using these settings some of the updates are inevitably lost and the

updates are not strictly real-time, but that is not a problem for this project - 50 Hz is plenty fast when the SLAM pipeline runs at 10-20 Hz a desktop computer.

2.3 ROS

The *Robot Operating System* is a framework that provides tools, libraries and conventions aiming to simplify the task of programming robots and robot systems. The libraries and tools available (many of which are open source) support a wide variety of robotic platforms and provide a plethora of capabilities. The flexibility and simplicity of ROS, along with support for the languages C++, Python, and Lisp has led to wide adoption of ROS within the robotics community.

ROS executables are called *nodes* and different nodes (which do not have to be on the same machine) can pass information to each other asynchronously over *topics* in the form of *messages*, via request-response *services*, or store and read data via a *parameter server*. The resulting network of nodes and their relations is called the *graph*.

ROS topics are updated when some node *publishes* a message to it and other nodes can then receive that message if they are *subscribed* to that topic. For example, the AR.Drone 2.0 used in this project sends information about its status and sensor readings (e.g., battery status, flying/landed, measured altitude, etc.) via the topic `/ardrone/navdata` by publishing messages of the type `ardrone_autonomy::Navdata` and receives navigation commands in the form of `geometry_msgs::Twist` messages by subscribing to the topic `/cmd_vel`. Most ROS communication in this project happens over topics, but during launch and setup some parameters on the parameter server are configured and a service is used to select which camera the drone should publish images from.

In this project ROS has been used for communicating with the drone and MATLAB has been used for running a SLAM algorithm using data received from the drone. The MATLAB *Robotics System Toolbox* [22] has been used to bridge the two systems.

2.3.1 ROS coordinate frames

Robot applications commonly require moving coordinates between different coordinate frames in order to orchestrate the robot's/robots' movements. To this end the ROS package *tf* makes it possible to keep track of multiple coordinate frames over time using a tree-like structure to relate the different coordinate systems. These coordinate frames are related via *coordinate frame transformations*, which are published at regular intervals. These transformations are translations and rotations that move points from one coordinate frame to another.

The AR.Drone 2.0 driver publishes coordinate frame transformations between four different coordinate frames: the camera coordinate systems of both the bottom and the front camera (`ardrone_base_bottomcam` and `ardrone_base_frontcam`, respectively); the local coordinate system of the drone, with its origin at the center, or *link*, of the drone (`ardrone_base_link`); and the global coordinate system within which the drone flies, which is tracked using on-board odometry (`odom`). The link/center coordinate system is the same one as the one shown in Figure 2.2b.

Chapter 3

Method

3.1 Problem formulation

A flying robot equipped with a downwards looking camera moves around in a plane ($z = z_{altitude} > 0$) parallel to another plane ($z = 0$), i.e., the floor. The camera is rigidly mounted to the drone and thus tilts with the drone as it moves around. A figure showing the problem geometry is shown in Figure 3.1. The position of the drone in the global coordinate system is described using the pose $\mathbf{c} = (t_x, t_y, \psi)$, where $(t_x, t_y, z_{altitude})$ is the robot's Cartesian coordinates in space and ψ is the robot's rotation around the z-axis (i.e., the *yaw*). Figure 3.2 shows how the pose parameters relate to the drone's position and heading in the global coordinate system.

In the ideal case the roll and pitch of the drone would always be zero so that the camera (and thus its optical axis) would always be perpendicular to the floor. Then any calibrated and zero-skew image taken with the camera would be a direct map of the imaged section of the floor, up to a scale factor. From such a local map of a subregion of the floor it would be possible to find the drone's pose $\mathbf{c}$ by finding a rigid transformation (a rotation around the z-axis and a translation in the plane) that registered the small local map to a bigger map of the floor in the global coordinate system.

In reality the optical axis was rarely perfectly perpendicular to the floor due to how the drone - and thus the camera - shifted as the drone maneuvered. Because of this there needed to be a way to compensate for the drone's roll

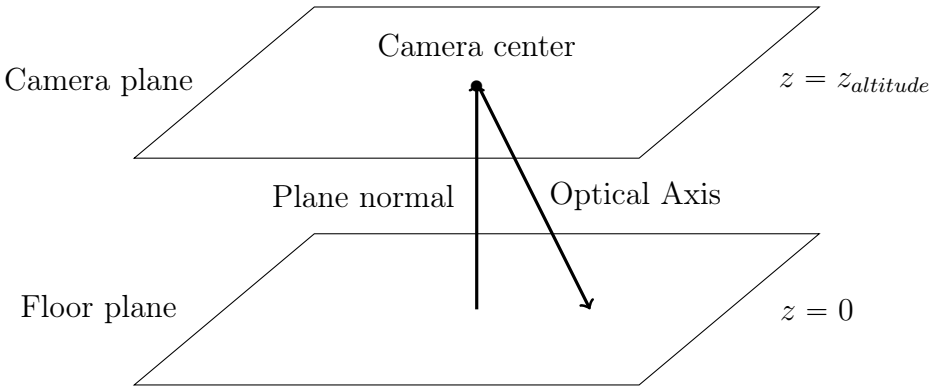


Figure 3.1: The problem geometry in the model. The camera moves around freely in the camera plane and takes images of the floor plane. The images are affected by the tilted optical axis and this needs to be corrected for.

and pitch - which were collectively labeled the *tilt* of the drone. A way to do this was by using a homography: if the camera's intrinsic parameters and its tilt were known then a homography could be constructed which mapped an image taken with the tilted camera to another image where the pitch and roll was compensated for, i.e., an image of the same scene, but appearing as if taken by a camera with no tilt. From there the pose of the drone could be determined in the same way as in the ideal case, by registering the local image/map to the global map.

Finally, some notes about the floor: because the algorithm relied on identifying and matching specific regions on the floor with each other there were some requirements on the texture of the floor. The floor could not be of a uniform color, and needed to feature details and patterns of reasonable scale that could be resolved by the drone's camera at the relevant altitude. "Reasonable scale" here meant that the details were not so small that they appeared as noise, and not so large that few or no features were fully contained within each given image. Most floors (e.g., wood, laminate, tiles of textured materials, etc.) fulfilled this requirement when imaged with standard resolution cameras at altitudes of a couple of decimeters to 1-2 meters, which was a reasonable altitude span for a quadcopter exploring an indoor environment.

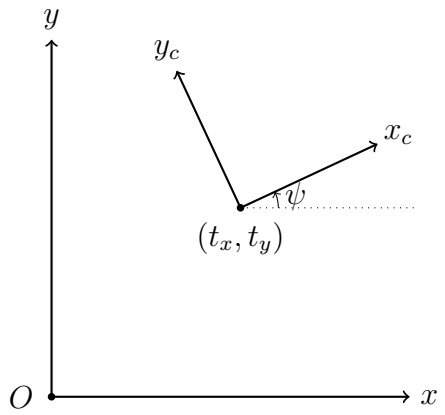


Figure 3.2: The two main coordinate systems of interest: x and y denotes the global x- and y-axes, x_c and y_c denotes the local x- and y-axes of the camera/drone, and $\mathbf{c} = (t_x, t_y, \psi)$ is the pose of the drone in the global coordinate system.

3.1.1 Relevant coordinate systems

A lot of different coordinate systems/frames were used in this project, so it is worth clarifying them and their relations to each other.

The *global coordinate system* is a rigid frame (it does not rotate or translate) where the navigation map is defined. The drone's relation to the global coordinate system is shown in Figure 3.2.

The *local coordinate system* is centered at the drone and rotates around the z-axis, but not the x- or y-axes, so that it is unaffected by the drone's tilt but tracks the drone's yaw and planar translation within the global coordinate system (again, see Figure 3.2). The relationship between the local and the global coordinate system is captured by the drone's pose, $\mathbf{c} = (t_x, t_y, \psi)$.

The *image coordinate system* has the unit "pixels" and is where the undistorted (but not rectified) images from the drone's bottom camera are defined. The images are affected both by the drone's tilt and the altitude of the drone (a scaling factor) - its relation to the local coordinate system is given by the rectification homography, $\mathbf{H}$, and the drone's altitude, $z_{altitude}$.

During the computation of the rectification homography (see Section 3.4.5) some additional coordinate frames are used. They are the drone's bottom *camera coordinate system* (which ROS tracks as the `ardrone_base_bottomcam` `tf`-frame); the drone's local, tilt-affected[1] *link coordinate system* (which ROS tracks as the `ardrone_base_link` `tf`-frame); and the drone's *odometry coordinate system* (which ROS tracks as the `odom` `tf`-frame). The drone's link and bottom camera coordinate systems have a constant relationship; the camera is rigidly mounted to the drone and does not move or rotate in relation to the drone's body. The relationship between the drone's link and odometry coordinate system is based on odometric measurements from the drone's onboard suite of sensors and tracks the drone's pose (rotation and translation) in relation to a fixed global coordinate system.

3.2 High-level design of the SLAM pipeline

The SLAM pipeline developed in this project could be coarsely divided into several steps. After an initial setup step the program entered a loop that could either be run indefinitely or for a set number of iterations. The steps in the loop are explained in more detail in the following sections, but a quick summary of the steps is listed below:

1. **Initial setup**: Setup the ARDrone and the required ROS nodes, load camera parameters and preallocate arrays for the map.
2. **Read data from the drone**: Capture the most recently published altitude, odometry and bottom camera image from the drone.
3. **Preprocess image**: Preprocess the image from the drone to improve contrast and remove lens distortion.
4. **Extract features**: Detect and extract SURF features from the preprocessed image.
5. **Update the rectification homography**: Update the rectification homography using the camera parameters and the latest odometry data.

[1] "Tilt-affected" here means that the coordinate system's x- and y-axes track the drone's roll and pitch, unlike the local coordinate system described in the previous paragraph.

6. **Match features**: Match the SURF features from the image to the features stored in the map.

7. **Rectify coordinates**: Rectify the coordinates of the matched features from the image using the homography.

8. **Estimate the drone's pose**: Estimate the drone's pose by finding a rotation and translation that registers the matched, rectified features from the last image to their matches in the map.

9. **Update the map**: Add novel features and the estimated pose to the map. Update metadata for the features in the map to reflect their detection from the new pose.

10. **Bundle adjustment**: Setup and run bundle adjustment on the matched points in the map and all poses associated with those points, using the constraints induced by the new measurements.

After step #10 the algorithm jumps back to step #2 and starts a new iteration.

3.3 ROS chessboard camera calibration

Before the pipeline was even started the parameters of the drone's camera needed to be known. To this end a process known as *geometric camera calibration*, *camera resectioning*, or simply *camera calibration* was used, whereby the parameters of the lens and image sensor of the camera were estimated. Specifically, the intrinsic parameters and the distortion coefficients were estimated using this process.

The method required taking multiple pictures of an object with easily identifiable features and known distances between those features, so that relations between image coordinates and world coordinates of the features could be established. The images had to to be from many different angles and distances so that all of the behaviors of the lens and image sensor were captured. Chessboards, with a known number of squares and known square dimensions, are commonly used as targets for camera calibration due to their high contrast and easy identification. For this project a chessboard with 10×7 squares

with 25 mm sides was used.

The technique is very common in vision and image applications, so most image processing toolboxes implement some form of camera resectioning tool. ROS was no exception and included a camera calibration package, `camera_calibration` [29], which was used for this project. This package was in turn based on OpenCV routines. Figure 3.3 shows a screenshot of the interface of the ROS camera calibrator tool.

After the calibration the estimated intrinsic camera parameters and distortion coefficients were sent to the drone. The drone then stored those parameters and loaded them whenever the drone booted and subsequently broadcasted them via the `/ardrone/camera_info` topic. The drone did not apply the calibration to the raw images before broadcasting them, instead the parameters were made available so that the user could choose how and if to undistort the images coming from the drone.

The intrinsic matrix for the bottom camera, according to the camera resectioning, was

$$\mathbf{K} = \begin{bmatrix} 684.41 & 0 & 295.23 \\ 0 & 687.49 & 179.05 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

Table 3.1 lists the lens distortion coefficients that were estimated by the camera calibration tool.

Table 3.1: Estimated values of the radial distortion coefficients, k_1 and k_2, and the tangential distortion coefficients, p_1 and p_2, for the drone's bottom camera. See Section 2.1.3 for an explanation of these coefficients.

Variable	k_1	k_2	p_1	p_2
Estimated value	0.143	-0.508	$-1.198 \cdot 10^{-3}$	$6.676 \cdot 10^{-4}$

Note that the lens distortion for the bottom camera was relatively small (especially the radial distortion) and that it thus was sufficient with only 2 (as opposed to 3) radial distortion coefficients. The small lens distortion was likely because the camera was designed to be used for optical flow ground speed measurements, and thus either needed to be calibrated and natively

undistorted (which would have put unnecessary strain on the drone's onboard CPU) or have low lens distortion to start with so that the raw images could be used to that end. The fact that the camera did not need to focus further reduced the need for a thick lens that could have added distortion.

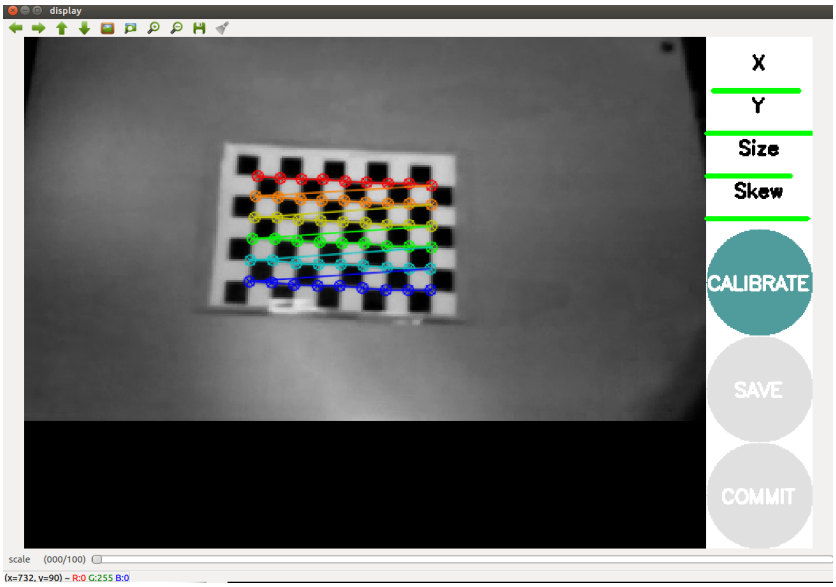


Figure 3.3: A screenshot of the ROS camera calibrator interface. Notice how the inner corners of the chessboard squares are identified and highlighted.

3.4 The SLAM pipeline

The SLAM pipeline was implemented in MATLAB (release R2017a). It relied heavily on the *Robotics System Toolbox* [22] for ROS integration and communication with the drone, and the *Computer Vision System Toolbox* [20] for feature detection and -extraction, camera calibration, and feature matching. The pipeline also required the ROS package `ardrone_autonomy` [25], which was used to launch the AR.Drone 2.0's driver ROS node.

The communication with the drone was handled via WiFi - the drone hosted a WiFi network over which it transmitted and received data.

3.4.1 Initial setup

Here the driver for the ARDrone was launched via the ROS node `ardrone_driver`, from the `ardrone_autonomy` package. The node was launched using a custom ROS `.launch` file which set a couple of ROS parameters for the drone: namely, it set the drone to transmit updates at 200 Hz; set the driver to publish received data from the drone via ROS messages at 50 Hz[2]; and set covariance values for the IMU (*Inertial Measurement Unit*)[3]. Then, a second ROS node, `matlab_node`, was launched, which was used to bridge the gap between ROS and MATLAB. Finally, some additional setup was performed: some more *drone setup*; reading *camera parameters* from the drone; and *preallocating arrays* for storing the the map.

For the *`matlab_node` setup* 4 ROS topic subscribers were attached to the MATLAB node. The topics subscribed to were: `/ardrone/image_raw`, where images from the camera were published; `/ardrone/camera_info`, where info about the camera, like the intrinsic parameters and the distortion parameters, were published; `/ardrone/navdata`, where various data about the drone's status, like its measured altitude, were published; and `/tf`, where coordinate frame transformations were published. In the MATLAB node setup some ROS topic publishers were also attached to the MATLAB node. These publishers were used to send navigation commands to the drone, via the ROS topics `/ardrone/takeoff`, `/ardrone/land`, and `/cmd_vel`, which handled commands for taking off, landing, and changing the velocity of the drone, respectively. The MATLAB integration of ROS allowed subscribers to have "callback functions", which were MATLAB functions that were called whenever the subscriber received a message. Each subscriber to the 4 topics was given a callback function that stored the last received message for that given topic in a MATLAB global structure array, `LATEST_MESSAGES`. That way the ROS messages published by the drone driver were converted into a data type that was easier to manage from within MATLAB.

The *drone setup* was simple: the only extra things that needed to be config-

[2]These update frequencies were the default values from the `ardrone_autonomy` package. They worked well enough so there was little reason to change them.

[3]Here, again, the default values from the `ardrone_autonomy` documentation were used. The three 3×3 covariance matrices for the IMU's linear acceleration, angular velocity and orientation, are listed in Appendix A.

ured on the drone at that point was which camera the drone should publish images from (front or bottom), and a "flat trim" request had to be sent to the drone, so that it zeroed its odometry[4] under the assumption that it was on a flat surface. The first was done via the `/ardrone/setcamchannel` service, and the second was done via the `/ardrone/flattrim` service.

During the flat trim-step the drone had to be landed on a horizontal surface parallel to the floor - otherwise, zeroing the pitch and roll estimates could ruin the homography construction, or, in the worst case, cause the drone to crash since it might have an incorrectly zeroed internal control system. If the odometry was not manually zeroed in this step the odometry would initialize dependent on the estimate of "down" from the accelerometer and the direction towards magnetic north from the magnetometer[5].

The *camera parameters* were read from the `/ardrone/camera_info` topic. The parameters that were required for the algorithm were the intrinsic camera parameters and the lens distortion coefficients. In addition to those camera parameters a coordinate frame transformation was also read in this step, via the `/tf` topic. The transformation was that from the bottom camera's coordinate frame (`ardrone_base_bottomcam`) to the center of the drone (`ardrone_base_link`). This transformation was constant due to the rigidity of the drone.

Finally, arrays for storing the map of features and poses were *preallocated*. For each SURF feature/landmark entered into the map, the map needed to hold the feature descriptor (of length 64), the feature's coordinates in the map (of length 2), and metadata describing each detection of that feature. That metadata was, for each detection of a given feature: the time step ("pose number") the feature was detected in, and the feature's coordinates in the local coordinate system of the pose it was detected from. This metadata was required for the bundle adjustment step later in the pipeline, where

[4]The ARDrone used its odometry readings to maintain an estimate of the drone's position and orientation in a fixed, global Cartesian coordinate system. Re-initializing the odometry in this way told the drone to update its odometry frame and use the current tilt as a "zero-tilt" reference. This feature was likely provided as a way to counter sensor drift - by allowing the user to "zero" the odometry readings.

[5]The standard outlined in ROS REP 105 aligns the x-axis east, y-north, and the z-axis up at the origin of the coordinate frame [24].

global feature coordinates and poses were optimized using data from all detections. For each pose/time step, the map needed to hold the estimated global pose of the drone (of length 3), and the coordinates of the principal point in the local coordinate system in that time step (of length 2). This altitude-compensated principal point was needed in order to separate the registration translation from the global pose of the drone for the given pose during bundle adjustment.

In the preallocation step assumptions were made about how many features would be detected over the drone's campaign, how many iterations it would go on for, and how many times a feature would be detected at the most. This assumption was of course scenario dependent and no global "best rule" could be found (or at least, finding such a rule was outside the scope of this thesis). The selected preallocation strategy assumed relatively short campaigns of ~ 500 iterations, seeing no more than 55000 features and up to 10 detections of each feature.

A pointer was used to keep track of the current size of the map within the preallocated arrays and should the map outgrow the arrays the arrays were re-preallocated and grown by 50%.

3.4.2 Read data from the drone

Once setup was complete the algorithm entered a typical SLAM loop [37] of landmark extraction, data association, state estimation, state- and landmark update. The first step was then to read data from the drone, via the ROS topics. The data required at the start of each iteration was the most recently published measured altitude, the coordinate frame transformation from the inertial odometry coordinate system (the frame `odom`) to the drone's link coordinate system (the frame `ardrone_base_link`), and the image from the bottom camera.

Figure 3.4a shows a typical image of the floor taken by the bottom camera from an altitude of about 75 cm. The image quality was not ideal; the raw image suffered from both upscaling and compression artifacts, see Figure 3.4b

(a) A typical raw image of the floor taken with the ARDrone 2.0 bottom camera. Note that the whole image is out of focus.

(b) Closeup of the raw image from Figure 3.4b, highlighting the compression- and upscaling artifacts. Note that the macroblocks from the compression are very apparent and that the resulting blockiness of the image bias the extracted features to be aligned with the x- or y-axis.

Figure 3.4: A raw image from the bottom camera.

3.4.3 Preprocess image

The raw image was preprocessed in order to aid with data extraction from the image further down the line. First, the image was converted to grayscale (SURF feature extraction only supports grayscale images) and then the

grayscale image was histogram equalized in order to improve contrast. After that the distortion parameters from the camera calibration were used to undistort the image, remapping image points to compensate for distortion caused by the lens of the camera.

Figure 3.5 shows the same image as in Figure 3.4, but after the preprocessing step.

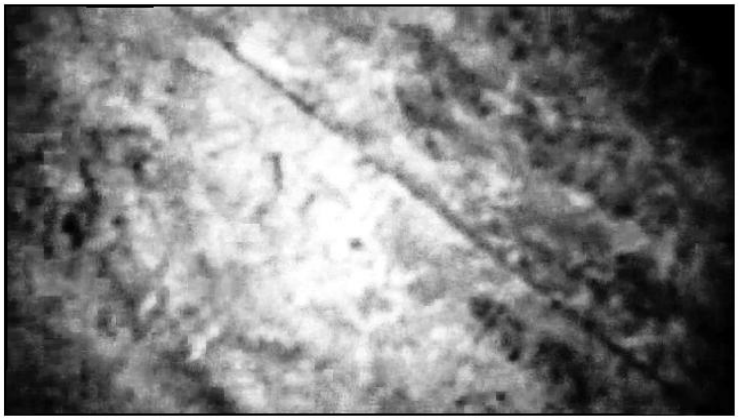


Figure 3.5: The same image from the bottom camera as in Figure 3.4, but after preprocessing. The camera's lens did not distort the image much, so the distortion correction is hard to notice. Some small black areas can be seen near the image's edges, caused by the image being "squeezed" together from the edges to counteract the (minor) radial distortion.

3.4.4 Extract features

Here SURF *features* were detected and feature *descriptors* extracted from the image, using the `detectSURFFeatures` and `extractFeatures` functions, respectively.

After detecting all SURF features in the image the (up to) 300 strongest detections were selected for feature descriptor(/vector) extraction. The strongest SURF features were selected using the `selectStrongest` method of the `SURFPoints` class in MATLAB, which culled SURF points based on the

strength of the response of the SURF feature detector. More distinct features (areas of more pronounced local change) gave stronger responses. The purpose of this culling of features was twofold: first, a smaller number of features was faster to compare with the map and in the long run also lead to a map with fewer features that was faster to search through, again improving the matching performance; second, because the images were so compressed and of such low quality the SURF detector occasionally classified artifacts in the image as features - rejecting weak features was one way of compensating for the bad image quality. In other words, this early feature/landmark culling was done both to increase performance and to increase robustness by removing "false" features/noise.

After the early feature culling the remaining features were used to extract feature vectors/descriptors from the image regions surrounding the features. A feature vector length of 64 was used - the alternative length of 128 was deemed too expensive, performance wise, and overkill, quality wise, given the application and image quality.

Figure 3.6 shows the pre-processed image from 3.5, with some of the strongest detected SURF features overlaid.

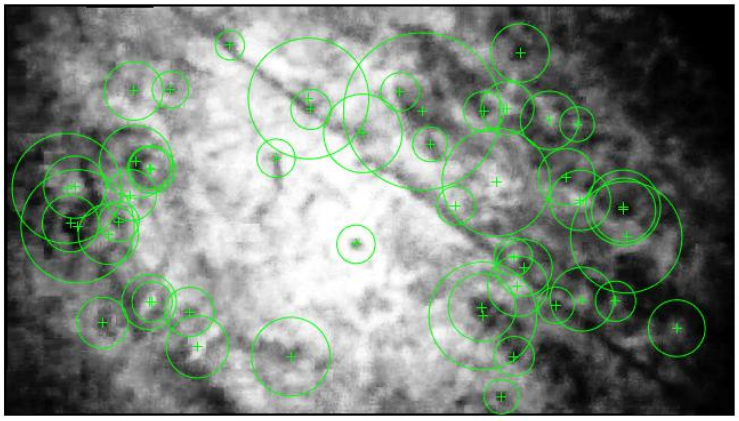


Figure 3.6: The preprocessed image from Figure 3.5, with the 50 strongest detected SURF features overlaid. The size of the green rings around each feature is proportional to the scale of the feature (at a ratio of $6 * \texttt{Scale}$).

3.4.5 Update the rectification homography

Here the homography $\mathbf{H}$, which was used to correct for the drone's tilt, was built using the measured pose of the drone in the same time step that the last image was taken in.

This homography was constructed using the intrinsic parameters of the camera (contained in the matrix $\mathbf{K}$), the ROS coordinate frame transformation from the bottom camera to the drone's center[6], and the ROS coordinate frame transformation from the drone's odometry coordinate system to the drone's center[7]. The first two of these were read and stored in the *Setup*-step of the pipeline and remained constant throughout the entire run; the camera never changed focus and the camera and drone center did not move relative to each other. The last one was read from the drone in the *Read data from the drone*-step.

The ROS coordinate frame transforms [30] contained both a translation, specified as a length 3 vector (x, y, z), and a rotation, specified as a quaternion (x, y, z, w). The transform was applied in the order *translation-rotation* ("TR"). The bottom-camera to drone-center transformation always had a translation of $(0, -0.02, 0)$ due to the bottom camera being situated near the back of the drone, 2 cm behind the drone-center.

The application of the tilt-correction homography onto a set of points in image space was a concatenation of a four-step operation:

1. The points in image space were moved to camera space using the inverse of the intrinsic camera matrix, $\mathbf{K}^{-1}$.

2. The points were translated -0.02 meters along the y-axis in camera space, using the matrix $\mathbf{T}$, essentially moving from "camera space" to "drone space".

3. The points were rotated around the drone to the locations they would have been in if the drone had no tilt, using a rotation matrix $\mathbf{R}_{\phi\theta}$.

[6] I.e., the ROS `tf` transformation from the `ardrone_base_bottomcam`-frame to the `ardrone_base_link`-frame.

[7] I.e., the ROS `tf` transformation from the `odom`-frame to the `ardrone_base_link`-frame.

4. The points were moved back to image space from the translated and rotated camera space by applying the intrinsic camera matrix $\mathbf{K}$.

Combining these four matrices give the tilt correction homography, $\mathbf{H}$:

$$\mathbf{H} = \mathbf{K}\mathbf{R}_{\phi\theta}\mathbf{T}\mathbf{K}^{-1} \tag{3.2}$$

Where the matrix $\mathbf{T}$ was given by

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -0.02 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

and the rotation matrix $\mathbf{R}_{\phi\theta}$ was in turn constructed from 3 other rotation matrices:

$$\mathbf{R}_{\phi\theta} = \mathbf{R}_{C2L}^{T}\mathbf{R}_{L2O}^{xy}\mathbf{R}_{C2L} \tag{3.4}$$

where $\mathbf{R}_{C2L}$ was the rotation from the camera frame (of reference) to the drone center frame (i.e., the "*link*" frame) and $\mathbf{R}_{L2O}^{xy}$ was the rotation from the drone center frame to the odometry frame's xy-plane, i.e., without rotating to align with the odometry's estimated heading.

The matrix $\mathbf{R}_{C2L}$ was easily derived by converting the rotation quaternion from the ROS coordinate frame transformation from the bottom camera to the drone center into a rotation matrix.

The matrix $\mathbf{R}_{L2O}^{xy}$ was harder to get as the rotation quaternion from the ROS coordinate frame transformation included a rotation to align the heading with the global frame. In order to extract only the pitch and roll corrections necessary to correct for the camera tilt the rotation described by the odometry-to-drone-center quaternion was converted to Euler angles[8], and the x- and y-components were extracted and used to construct a rotation matrix, $\mathbf{R}_{O2L}^{xy}$, which described the pitch and roll of the drone in the odometry's frame of reference. Transposing that rotation matrix then yielded a rotation matrix describing the pitch and roll that needed to be applied to the drone in order to align it with the odometry plane (which was, excluding sensor errors, parallel to the floor plane), i.e.,

$$\mathbf{R}_{L2O}^{xy} = (\mathbf{R}_{O2L}^{xy})^{T} \tag{3.5}$$

[8]In the sequence "ZYX".

The reason behind using the rotation from the odometry frame to the drone center frame, extracting the pitch and roll rotations and then transposing the resulting rotation matrix to get a pitch- and roll correcting rotation matrix from the drone center frame to the odometry/floor frame had to do with the Euler rotation sequences supported in MATLAB. MATLAB (as of R2017a) only supported "ZYX" and "ZYZ" sequences of Euler angles, but had it supported the "XYZ"-order it would have been possible to use the drone-center-to-odometry transformation and forgo the the final transposition.

Map initialization

In the first iteration, when initializing the map, the rectification homography was constructed and applied to the feature coordinates in the same manner as in the later iterations (the application of the homography is discussed in Section 3.4.7). Also, the measured tilt of the quadcopter in each iteration was not a variable that was stored and optimized later (unlike the estimated pose of the drone and the coordinates in the map of individual features). This meant that the feature coordinates were not rectified to some sort of persistent plane, but rather the rectification was done on a per-iteration basis. Specifically, the rectification in each iteration (including the first) was done to a plane perpendicular to whatever direction the drone's sensors thought was "down" (gravity-wise) at that point in time.

This approach was a bit naïve because it assumed that the odometry readings used to construct the odometry were always good (enough) and that sensor drift didn't happen or was negligible. Still, it worked well enough and adding the drone's tilt in each iteration as a variable to be optimized during the bundle adjustment step would have complicated the pipeline beyond the scope of this project.

3.4.6 Match features

The feature matching step was the most performance intensive step of the algorithm due to the large number of features that needs to be compared, especially when the map was big, e.g., after the SLAM loop had run for multiple iterations. The matching of newly detected SURF feature descriptors/vectors to previously detected feature vectors stored in the "map" could be done in different ways. The most robust way was brute force matching

between all new feature vectors and all previously existing feature vectors in the map, but this became become prohibitively slow as the map grew larger.

Fast feature matching via geometric distance culling

In order to improve matching speed as the map grew the algorithm first tried to find matches within a subsection of the map, selected by filtering out features that were too far away from the last estimated position of the drone. The culling step still had to "touch" each entry in the map, as it needed to look at the (x, y) coordinates of each feature, but operating on length 2 vectors is (in general) faster than operating on length 64 vectors.

If this matching step with the smaller, culled map failed (too few matches were found) - or if the previous iteration failed to estimate the pose of the drone - then a full brute force matching with the whole map was performed. This provided a failsafe should the drone move very far between two updates, or if the estimated location in one step was very wrong, or if the algorithm failed outright in one iteration.

The culling was done by identifying all features in the map where

$$\|(x_f, y_f) - (t_x, t_y)_{i-1}\|_2 < d_{max} \tag{3.6}$$

where (x_f, y_f) were the coordinates of the feature, $(t_x, t_y)_{i-1}$ the estimated coordinates of the drone in the previous iteration, and d_{max} was a cutoff threshold. A maximum distance of 350 pixels was selected for this threshold as it gave a good performance increase without noticeably compromising the robustness of the algorithm.

Feature matching

In both the cases of feature matching (against the full map and against a smaller map as part of the sped-up matching subroutine) the MATLAB function `matchFeatures`[21] was used to match SURF feature descriptors from the last image with features in the map. The function tried to find matching feature vectors between the two sets by finding vectors that were close to each other, using the sum of squared differences metric while also applying some other criteria to provide good matches. Some special settings were used: unique matches were enforced, and the "max ratio" (which is explained

below) was set to 0.5, as opposed to the default 0.6.

Forcing matches to be unique meant that each feature vector in the map could only be matched to one feature vector from the most recent image, and vice versa. This helped remove ambiguous matches as only the strongest match to an ambiguous feature vector was kept.

The lower max ratio (which was MATLAB's name for "Lowe's Ratio", from [17]) also helped with filtering out generic/ambiguous matches by setting a maximum allowed ratio between the distance to the closest match and the next closest match. If the nearest neighbor to a feature vector was both within the "matching threshold"[9] and the ratio between the distance to that nearest neighbor and the distance to the next-nearest neighbor was less than 0.5 then the nearest neighbor was deemed a match to the given feature vector. In other words: Lowe's ratio filtered out matches where other potential matches were not significantly worse matches than the best match. A lower threshold forced matched features to be more unique/descriptive, as generic and/or false matches were likely to have other generic or false matches within a similar distance.

Bad match removal ("Rudbeck's Algorithm")

In order to further filter out false matches the matches returned by `matchFeatures` were filtered once more, this time with the goal of removing matched pairs that were deemed too different from each other. This was done using an algorithm first implemented by Rudbeck in [38]. The idea was to filter out matched feature pairs dependent on the distance between them using a variable threshold.

As mentioned before, feature vectors were compared by the sum of squared differences distance between them, which for two vectors $\mathbf{p}$ and $\mathbf{q}$ of length k is defined as

$$d(\mathbf{p}, \mathbf{q}) = \sum_{i=1}^{k} (p_i - q_i)^2 \tag{3.7}$$

Given a set of matched feature vectors, the smallest distance between any

[9] The default value of 1.0, i.e., "the distance had to be less than 1% away from the distance of a perfect match", was deemed sufficient.

matched pair in the set was denoted d_{min}. Then, d^*_{min} was defined as

$$d^*_{min} = max(d_{min}, 0.02^2) = max(d_{min}, 0.0004) \tag{3.8}$$

This thresholded smallest distance was then used to remove any matches where

$$d > 2d^*_{min} \tag{3.9}$$

This put an upper bound on how different matches could be, using a threshold that, to some extent, adapted to how good the matches were in the given set. The thresholding of d_{min} into d^*_{min} was done to prevent a particularly good match from wiping out matches that would otherwise have been sufficiently good.

The threshold of 0.02^2 and the multiplier of 2 were taken directly from [38]. Some experiments were done with different values but in the end Rudbeck's values were deemed to be suitable for this application.

Because the pose estimation later in the pipeline required at least two point correspondences to work the algorithm failed the current iteration if there were fewer than two matches remaining after this match culling step. If that happened the "estimated" pose of the current iteration was set to the same pose as the last iteration, and then the current iteration was exited (i.e., the map was not updated and no bundle adjustment was done) and the fast feature matching algorithm was disabled for the next iteration. The last step there was done to ensure that the next iteration used the "fallback" full map matching method.

First iteration special case

The first iteration of the pipeline was a special case, because then the map was empty and there were no previously observed landmarks to match the newly observed landmarks with. The solution chosen here was to simply classify all newly observed features as matches to the (empty) map and then moving on to the next step in the pipeline.

3.4.7 Rectify coordinates

While it is possible to apply a homography to a complete image it is faster and simpler to just apply the homography to the local (pixel) coordinates of

the features from the image.

The unrectified coordinates of a feature in image space, $\mathbf{X} = (x, y, 1)$, and its rectified/tilt corrected coordinates, $\mathbf{X}^{\perp} = (x^{\perp}, y^{\perp}, 1)$, are related via the rectification homography $\mathbf{H}$:

$$\lambda \mathbf{X}^{\perp} = \mathbf{H}\mathbf{X} = \mathbf{K}\mathbf{R}_{\phi\theta}\mathbf{T}\mathbf{K}^{-1}\mathbf{X} \tag{3.10}$$

where λ is the depth of the rectified coordinate after applying the homography, i.e., it is necessary to perform a depth-division on the coordinates after applying the homography in order to arrive at an image where the points appear as if they were taken with a tilt-free camera.

This placed the rectified points in a plane at depth 1 in image space, but the variable altitude of the drone needed to be accounted for as well. The floor area covered by one pixel varied depending on the altitude of the drone, and hence some additional scaling needed to be applied to correct for this. So, the final step of the coordinate rectification step was to scale all feature coordinates with the measured altitude of the drone, z_{alt}, placing them in a plane at a depth directly proportional to the altitude of the camera[10].

Ideally, the camera should stay at a constant height while exploring, but due to how it could tilt and shift about as it flew around this correction was necessary, not to mention that it allowed the drone to be more flexible as it navigated the environment, e.g., by flying lower to avoid an overhead obstacle.

The local camera pose

The *principal point*, (x_0, y_0), - which can be extracted from the intrinsic camera matrix - represents the location of the camera center within the image coordinate system (recall that the principal point represents an offset for the point $(0, 0)$ when moving from the camera coordinate system to the image coordinate system).

[10]Here meters were chosen as the unit of the measured altitude. The unit of the altitude measurements published by the drone was millimeters, but scaling with meters provided more readable axes when plotting a map of feature coordinates and robot poses.

The principal point was specified in pixels, and in order to remain consistent with the other points in the local coordinate system it also needed to be scaled dependent on the drone's altitude. Thus, the same scaling factor, z_{alt}, that was applied to the rectified and depth-divided feature coordinates was also applied to the principal point.

$$(x_0^r, y_0^r) = z_{alt} \cdot (x_0, y_0) \tag{3.11}$$

This scaled principal point then represents the coordinates of the camera in the local coordinate system of the drone.

Note that because it was possible for the altitude to fluctuate between iterations it was necessary to store the scaled principal point (or the measured altitude) for each iteration in order to separate the drone's local position in each iteration from the translation needed to align that image with the map. This separation was required in order to optimize the pose during the bundle adjustment step.

3.4.8 Estimate the drone's pose

The pose of the drone in each iteration consisted of two parts: a rotation and a translation, i.e., the pose described a rigid transformation. The most plausible transformation estimate was the one that did the best job of aligning the matched features from the most recent image with their corresponding matches in the (global coordinates) map when applying the transformation to the local coordinates of the re-detected landmarks. In other words, estimating the pose of the drone based on the matched features could be seen as a *image registration* problem, where the goal is to find a transformation that aligns one image with another image (or with a subregion thereof).

A rotation and translation in 2D can be estimated by using only two point correspondences, and while it was possible to estimate using more than two correspondences via least-square fitting this was opted against. Instead, a RANSAC approach was chosen to further filter out and reduce the impact of bad feature matches/outliers.

Least square fitting of R and t

Given a set of n point correspondences the best rigid transformation that aligns the local point coordinates, $\mathbf{X}_i$, with the global point coordinates, $\mathbf{x}_i$, is the $\mathbf{R}$ and $\mathbf{t}$ that solves the non-linear least-squares minimization problem

$$\begin{aligned} &\min_{\mathbf{R},\,\mathbf{t}} \sum_{i=1}^{n} \|\mathbf{x}_i - (\mathbf{R}\mathbf{X}_i + \mathbf{t})\|^2 \\ &\text{s.t. } \mathbf{R}^T\mathbf{R} = \mathbf{I}\,,\ \det(\mathbf{R}) = 1 \end{aligned} \tag{3.12}$$

There are many possible ways to solve this problem, e.g., using eigenvalue-eigenvector decompositions [12], singular value decompositions (SVDs) [2], or by using Lagrange multipliers [5, 33].

The method chosen for this project was the method using SVDs, outlined in [2], which was also the method used by Brange and Rudbeck in their papers. This method, given n pairs of corresponding points, $\mathbf{x}_i$ and $\mathbf{X}_i$, is described below:

First, find and subtract away the centroids, $\bar{\mathbf{x}}$ and $\bar{\mathbf{X}}$, of each of the two sets

$$\begin{aligned} \bar{\mathbf{x}} &= \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i \\ \bar{\mathbf{X}} &= \frac{1}{n}\sum_{i=1}^{n} \mathbf{X}_i \end{aligned} \tag{3.13}$$

The new, centered point coordinates are then

$$\begin{aligned} \hat{\mathbf{x}}_i &= \mathbf{x}_i - \bar{\mathbf{x}} \\ \hat{\mathbf{X}}_i &= \mathbf{X}_i - \bar{\mathbf{X}} \end{aligned} \tag{3.14}$$

Then calculate the sum of pairwise outer products matrix $\mathbf{C}$

$$\mathbf{C} = \sum_{i=1}^{n} \hat{\mathbf{X}}_i \hat{\mathbf{x}}_i^T \tag{3.15}$$

Find the singular value decomposition of $\mathbf{C}$

$$\mathbf{C} = \mathbf{U}\mathbf{S}\mathbf{V}^T \tag{3.16}$$

The rotation $\mathbf{R}$ and translation $\mathbf{t}$ that minimizes (3.12) is then given by

$$\mathbf{R} = \mathbf{U} \begin{bmatrix} 1 & 0 \\ 0 & \det(\mathbf{U}\mathbf{V}^T) \end{bmatrix} \mathbf{V}^T \tag{3.17}$$

$$\mathbf{t} = \bar{\mathbf{x}} - \mathbf{R}\bar{\mathbf{X}} \tag{3.18}$$

Image registration with RANSAC

In each iteration of the RANSAC method two matches were used to estimate the rotation and translation for registering the image, then all the matches were transformed using the newly estimated rigid transformation and the number of inlier points were counted. After all the RANSAC iterations were completed the transformation that resulted in the most inliers was selected. Then, a final check was performed to see if a sufficiently large fraction of all the matches were classified as inliers using the estimated best transformation[11] - this was for the case where even the best registration was "bad". In that case the estimated drone pose was rejected and the algorithm "failed" that iteration, leaving the estimated pose unchanged from the previous iteration and then the current iteration was exited in the same way as if too few matches were found in the "Match features"-step of the pipeline.

This failure could happen for example when a lot of false matches made it through the previous culling steps or when the tilt correction homography was wrong (due to bad odometry in the given step) so that no rigid transformation could properly register the image to the map.

The whole robust RANSAC image registration algorithm is outlined below:

1. At the start of each RANSAC iteration, select 2 pairs of point correspondences at random.
2. Find the $\mathbf{R}^*$ and $\mathbf{t}^*$ that solves (3.12) given the two matched pairs.
3. Transform all image feature points using $\mathbf{R}^*$ and $\mathbf{t}^*$.
4. Count the number of inliers. A point is counted as an inlier if the transformed point from the image is closer than 10 pixels to its match in the map.

[11] In the implementation at least 40 % of the matches needed to be inliers.

5. Repeat steps 1-4 for a set number of iterations.

6. Select the rotation $\mathbf{R}$ and translation $\mathbf{t}$ that resulted in the largest number of inliers. If there is a tie select the tied transformation that did the best job of minimizing (3.12).

The drone's pose

The transformation that registered the most recent image with the map translated and rotated all the points within the image, including the principal point, which represented the camera center. In order to get the drone's pose in the global map, $\mathbf{c} = (t_x, t_x, \psi)$, the registration transformation was applied to the scaled principal point (x_0^r, y_0^r) and the rotation angle ψ was extracted from $\mathbf{R}$:

$$(t_x, t_y) = \mathbf{R} \cdot (x_0^r, y_0^r) + \mathbf{t} \tag{3.19}$$

$$\mathbf{R} = \begin{bmatrix} \cos(\psi) & -\sin(\psi) \\ \sin(\psi) & \cos(\psi) \end{bmatrix} \implies \psi = \arctan2(\sin(\psi), \cos(\psi)) \tag{3.20}$$

First iteration special case

The first iteration of the SLAM algorithm is a special case because there are no matches in the map to register the features from the image to. Instead, all features and the principal point are transformed with the translation $\mathbf{t} = (0, 0)$ and the rotation angle $\psi = 0$, giving the initial pose $(x_0^r, y_0^r, 0)$.

3.4.9 Update the map

If a transformation registering the image to the map was found the map was updated by adding entries for all first-detection landmarks/features[12] and updating the map metadata to include information about all the features detected in the current iteration.

Each first-detection landmark had its length 64 feature vector and its length 2 coordinates added to the map.

[12] "First-detection features" are features that were not matched to another feature in the map. Features that were matched to a feature in the map are referred to as "re-detected features".

The map metadata included information about each detection of each feature - this information was required in the bundle adjustment step. The metadata about each feature included which iterations the feature had been detected in and the local, rectified coordinates of the feature in that given iteration. Such a metadata entry was added for each feature detected in the iteration, both first-detection and re-detected features.

Thus, each feature/landmark stored in the map required storing a length 64 single precision vector, a length 2 single precision vector, and a number of length 3 single precision vectors depending on how many times that landmark had been detected. If the map was sufficiently preallocated the map updates are relatively cheap and effective. As was mentioned previously, the map was preallocated in order to fit a generous amount of entries, and if the map outgrew the preallocated size it was re-allocated with 50 % more available entries. The metadata entries were preallocated in the same way, allowing many feature re-observations before the metadata array had to be reallocated.

First iteration special case

In the first iteration all observed landmarks from the image were first-detection features and they were added to the map accordingly.

3.4.10 Bundle adjustment

As the last step of each iteration of the SLAM pipeline a bundle adjustment routine was launched if the pose estimation was successful. The new observations of the re-detected features were used to refine the models of the coordinates of those features in the map and of all the poses from which those features have been observed.

The inspiration for this implementation of bundle adjustment came from a paper on sparse bundle adjustment for 2D mapping [13], which was also utilized by Brange [4].

As the local coordinates of re-observed features are registered to the global

coordinates there will inevitably[13] be some *reprojection errors*, where the transformed coordinates do not perfectly line up with the coordinates in the map. Given a match/re-observed feature with coordinates in the map $\mathbf{x} = (x, y)$ and local (rectified) coordinates $\mathbf{X} = (X, Y)$, which were registered to the map using the 2×2 rotation matrix $\mathbf{R} = \begin{bmatrix} \mathbf{R}^x \\ \mathbf{R}^y \end{bmatrix}$ and the translation vector $\mathbf{t} = (t^x, t^y)$ the reprojection error for that feature and estimated pose is given by

$$e = \|(e^x, e^y)\|^2 = \left\| \left(x - \left(\mathbf{R}^x \mathbf{X} + t^x \right), \, y - \left(\mathbf{R}^y \mathbf{X} + t^y \right) \right) \right\|^2 \tag{3.21}$$

For multiple features being detected from multiple poses each detection of each feature is associated with its own reprojection error; feature i being detected from pose j is associated with the error e_{ij}.

The generalization of (3.21) to get the sum of all reprojection errors ("total error"), given n features and m_i detections of feature i, is given by

$$\begin{aligned} e_{tot} = \sum_{i=1}^{n} \sum_{j=1}^{m_i} e_{ij} &= \sum_{i=1}^{n} \sum_{j=1}^{m_i} \|\mathbf{e}_{ij}\|^2 = \sum_{i=1}^{n} \sum_{j=1}^{m_i} \|(e_{ij}^x, e_{ij}^y)\|^2 \\ &= \sum_{i=1}^{n} \sum_{j=1}^{m_i} \left\| \left(x_i - \left(\mathbf{R}_j^x \mathbf{X}_{ij} + t_j^x \right), \, y_i - \left(\mathbf{R}_j^y \mathbf{X}_{ij} + t_j^y \right) \right) \right\|^2 \end{aligned} \tag{3.22}$$

where (x_i, y_i) are the coordinates of feature i in the map; $\mathbf{R}_j = \begin{bmatrix} \mathbf{R}_j^x \\ \mathbf{R}_j^y \end{bmatrix}$ is the rotation matrix corresponding to the estimated pose angle ψ_j of pose j; $\mathbf{X}_{ij}$ are the local coordinates of feature i observed from pose j; and (t_j^x, t_j^y) is the estimated translation vector associated with pose j.

The error function (3.22) is the objective function that the bundle adjustment algorithm tries to minimize. This is a non-linear least-squares problem, and in order to find a local minimum to this the *Levenberg-Marquardt* algorithm is used.

[13] These errors stem from many factors, such as quantization errors due to the discrete resolution of digital images and image compression, or from non-ideal registrations due to outliers affecting the pose estimation.

First, in order to apply the Levenberg-Marquardt method the rotation matrices $\mathbf{R}_j$ need to be linearized. This can be done via the exponential map [32]:

$$\exp(\mathbf{A}) = \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{A}^k = \mathbf{I} + \mathbf{A} + \frac{1}{2}\mathbf{A}^2 + \frac{1}{6}\mathbf{A}^3 + \dots \tag{3.23}$$

Given a rotation estimate, $\mathbf{R}_0$, any other rotation $\mathbf{R}$ can be written as $\mathbf{R}_0$ multiplied with the exponential map of a skew symmetric matrix

$$\mathbf{R} = \exp\begin{pmatrix} 0 & -a \\ a & 0 \end{pmatrix}\mathbf{R}_0 \approx \left(\mathbf{I} + \begin{bmatrix} 0 & -a \\ a & 0 \end{bmatrix}\right)\mathbf{R}_0 = (\mathbf{I} + a\mathbf{S})\mathbf{R}_0 \tag{3.24}$$

where only the first order terms of (3.23) were considered for the purposes of linearization and where $\mathbf{S} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$. The equation

$$\mathbf{R} \approx \mathbf{R}^{lin} = (\mathbf{I} + a\mathbf{S})\,\mathbf{R}_0 = \begin{bmatrix} 1 & -a \\ a & 1 \end{bmatrix}\mathbf{R}_0 \tag{3.25}$$

is then a linear local parameterization of the rotation $\mathbf{R}_0$ in the variable a. Each term in the sum of the objective function (3.22) is thus - after local linearization - associated with the variables $\mathbf{v}_{ij} = (x_i, y_i, t_j^x, t_j^y, a_j)$ and the vector function $\mathbf{e}_{ij} = (e_{ij}^x, e_{ij}^y)$. The corresponding partial derivatives $\frac{\partial \mathbf{e}_{ij}}{\partial \mathbf{v}_{ij}}$ are

$$\begin{aligned}
&\frac{\partial e_{ij}^x}{\partial x_i} = 1 && \frac{\partial e_{ij}^y}{\partial x_i} = 0 \\
&\frac{\partial e_{ij}^x}{\partial y_i} = 0 && \frac{\partial e_{ij}^y}{\partial y_i} = 1 \\
&\frac{\partial e_{ij}^x}{\partial t_j^x} = -1 && \frac{\partial e_{ij}^y}{\partial t_j^x} = 0 \\
&\frac{\partial e_{ij}^x}{\partial t_j^y} = 0 && \frac{\partial e_{ij}^y}{\partial t_j^y} = -1 \\
&\frac{\partial e_{ij}^x}{\partial a_j} = \mathbf{R}_{0j}^y \mathbf{X}_{ij} && \frac{\partial e_{ij}^y}{\partial a_j} = -\mathbf{R}_{0j}^x \mathbf{X}_{ij}
\end{aligned} \tag{3.26}$$

The Levenberg-Marquardt update

In each iteration of the bundle adjustment algorithm the following Levenberg-Marquardt (LM) update (which here includes a precision matrix to add weight to the errors) was computed

$$\Delta \mathbf{v} = -\big(\mathbf{H} + \lambda \operatorname{diag} \mathbf{H}\big)^{-1} \mathbf{J}^T \mathbf{\Lambda}\, \mathbf{e} \tag{3.27}$$

where

$$\mathbf{\Lambda} \equiv \begin{bmatrix} \Lambda & & \\ & \ddots & \\ & & \Lambda \end{bmatrix} \tag{3.28}$$

$$\mathbf{J} \equiv \frac{\partial \mathbf{e}}{\partial \mathbf{v}} \tag{3.29}$$

$$\mathbf{H} \equiv \mathbf{J}^T \mathbf{\Lambda} \mathbf{J} \tag{3.30}$$

and where, in turn, Λ was a precision matrix w.r.t. the errors in the x- and y-dimensions[14] and $\mathbf{e}$ was a vector containing all the error residual vectors $\mathbf{e}_{ij}$. The Jakobian (3.29) was filled in using the partial derivatives (3.26), iterating over all relevant features i and poses j.

The variables $\mathbf{v}$ were then updated using $\mathbf{v}_{new} = \mathbf{v}_{old} + \Delta\, \mathbf{v}$ and the objective function was re-evaluated as $e_{tot}(\mathbf{v}_{new})$. If the update resulted in a smaller value for the objective function the new variables were kept, and then a new iteration was started, unless some criterion was fulfilled that signified that the algorithm was done. After each LM-update the resulting $\mathbf{R}_j^{lin}$ (3.25) were normalized before re-evaluating the objective function (3.22) and before updating $\mathbf{J}$, if the update was successful.

The first pose

The pose from the first iteration of the pipeline was been defined to be associated with a rotation of 0 degrees and a translation of $(0, 0)$ pixels. Because of this the variables of the first pose, i.e., (t_1^x, t_1^y, a_1), were not included as

[14]The precision matrix used in this project was derived by measuring the average re-projection errors for the x- and y-dimensions over a large amount of iterations, forming a 2×2 covariance matrix and inverting it to get the precision matrix.

variables in the bundle adjustment algorithm, meaning that each detection of a feature i from the first pose was only associated with the partial derivatives

$$\begin{aligned} \frac{\partial e_{i1}^x}{\partial x_i} &= 1 \quad \frac{\partial e_{i1}^y}{\partial x_i} = 0 \\ \frac{\partial e_{i1}^x}{\partial y_i} &= 0 \quad \frac{\partial e_{i1}^y}{\partial y_i} = 1 \end{aligned} \tag{3.31}$$

The reprojection errors for these detections were still calculated the usual way, i.e., using (3.21) and the rotation $\mathbf{R}_1 = \mathbf{I}$ and the translation $\mathbf{t} = \mathbf{0}$.

Implementation in code

The map metadata - which included information about each detection of each feature in the map - was used to find all the m_i poses relevant to each of the n re-detected inlier[15] features and the rectified image coordinates, $\mathbf{X}_{ij}$, of each feature in each respective detection. The rotation $\mathbf{R}_{0j}$ relevant to each pose was calculated from the pose angle ψ_j, and the registration translation $\mathbf{t}_j = (t_j^x, t_j^y)$ was reconstructed from the pose coordinates by solving (3.19) for $\mathbf{t}$ using $\mathbf{R}_{0j}$ from above and the rectified principal point for iteration j.

Bundle adjustment using Levenberg-Marquardt as outlined above was then run until the decrease in the total reprojection error reached a plateau or for at most 10 iterations[16]. The error decrease was said to have plateaued if there was a successful update (error decreased) which yielded a less than 5 % error reduction. After the Levenberg-Marquardt loop exited the poses and the feature coordinates in the map were updated to the new, optimized values. If any of the updated poses were associated with some features in the map that had only been detected from that pose then those features were also updated to reflect the new pose.

An optimal initial value of λ (see (3.27)) and how to best update it is application specific, but for this application an initial value of $\lambda = 10$ and a scaling factor of 3 was deemed to be a good fit. If an update was successful in decreasing the total reprojection error the current value of λ was divided by 3, otherwise λ was multiplied by 3.

[15] Including outlier features in the bundle adjustment step would wreak havoc on the results, due to most of them being false matches.

[16] This was chosen empirically as a balance between error reduction and performance.

Chapter 4

Results

The experiments were done in the Robotics Lab on LTH, using an AR.Drone 2.0 (see Section 2.2) flying over a plastic laminate floor. The drone was maneuvered around using scripts with specific timings for the movement commands in order to try to get as repeatable movements as possible. It was deemed infeasible to get ground truth against which the live runs could be compared[1], so the mapping accuracy of the pipeline had to be evaluated using more qualitative methods (as opposed to quantitatively measuring errors between the estimated path and the true path, or between the estimated map and the true map).

The experimental data was obtained by commanding the drone to fly in squares, at about 75 cm altitude and at a velocity of 0.1 m/s, and then comparing the resulting estimated map and poses with the observed flight path to see if it was reasonable and continuous. The drone's maneuvers were not particularly precise: the drone rarely moved in a perfect square and instead moved in some sort of open ended quadrilateral, the drone rarely managed to keep its position perfectly steady while hovering and instead drifted around, and the drone often shifted/wobbled before or after maneuver commands were received. Still, the resulting path was more or less a square and experiments could always be repeated until acceptable paths were obtained.

[1]The drone, mainly being marketed towards consumers, was not capable of making small, precise maneuvers with good repeatability (unlike the robot used by Rudbeck [38]) and mounting the drone in some rig and manually moving it around was deemed overkill, out of scope, and too far removed from the realistic scenario of the drone flying around at speed.

Both higher and lower altitudes than 75 cm were tested briefly, but it was found that it made little difference to the results, so drone altitude was not used as a variable during the experiments. Different velocities than 0.1 m/s were also tested; it was quickly discovered that velocity commands of *less than* 0.1 m/s led to erratic and unpredictable maneuvers and that velocity commands of *more than* 0.1 m/s made the pipeline less stable. The drone was not very good at flying slowly; when given slow velocity commands (< 0.1 m/s) there would sometimes be a delay before the drone started moving in the commanded direction, and sometimes entire movement commands would be ignored (e.g., instead of strafing forward, left, back and then right then drone could just wait, then strafe left, back and right). The higher instability of the pipeline when the drone was given fast velocity commands (> 0.1 m/s) was due to significant motion blur in the images at those velocities. Motion blur was already an occasional problem at 0.1 m/s, but at higher velocities that became an insurmountable problem. Figure 4.1 shows two images from the bottom camera taken at different velocities, showing the motion blur that came with higher velocities.

(a) Image taken when drone moved at 0.1 m/s

(b) Image taken when drone moved at 0.2 m/s

Figure 4.1: Two images captured at different drone velocities, showing the the impact drone velocity has in terms of motion blur. Notice that the image taken at the higher velocity has noticeably less detail. Also notice that the color in the images differ, despite the images being taken under seemingly identical lighting conditions (the explanation for this difference is unknown).

Lighting conditions weren't initially considered a significant factor, but it was quickly discovered that the camera was very sensitive to changes in illumination - it was important for the results that the floor was very well and consistently lit. The lighting sensitivity was so severe that there was

a noticeable difference in algorithm robustness between experiments using summer daylight for illumination and experiments using the ceiling lights for illumination. Summer daylight was preferable, likely because it resulted in less glare via the shiny plastic laminate floor and gave a more even, diffuse light than the strong fluorescent ceiling lights did.

During the experiments many of the drone's maneuver campaigns were recorded so that they could be examined in more detail later. The recordings were done by sampling the drone's ROS topics that the MATLAB node subscribed to (`/ardrone/image_raw`, `/ardrone/camera_info`, `/ardrone/navdata` and `/tf`) at a high frequency (20 Hz) and storing the received messages so that they could be re-published later by a *dummy drone*. The dummy drone was written in MATLAB and mimicked the real drone by publishing such recorded messages at the same rate which they were sampled at to identically named topics as those of the real drone. These dummy recordings were used to study various parts and aspects of the pipeline on repeatable datasets. Such recordings were also used heavily during development of the pipeline in order to streamline the process by eliminating the need to go to the Robotics Lab to test new features and changes.

4.1 Pose estimation robustness

The robustness of the pipeline was very inconsistent. During the development phase - using saved campaign data from an early test run - the algorithm was relatively robust and the drone's estimated path was a reasonable estimation of the true path. The results during the experiment phase were drastically different: there the pipeline was completely unreliable and not a single successful run could be obtained. This was the case both for live runs and offline runs using data saved from live runs.

The early test run - which led to successful runs during the development - was recorded under different circumstances than the later runs in the experiment phase. The test run was recorded before the drone maneuver scripts was written, so instead the drone was manually carried around at a height of about 75 cm. Because of this (and the inability of the drone to reliably fly slowly, described in the previous section) the drone moved around at a lower speed than what it did during the experiment phase, leading to less motion

blur in the images. The lighting conditions was also different during the the test run: the test run was recorded during June on a day with with good daylight, and the experiments were done in November/December on days with overcast weather or after sunset. During the experiments the strong fluorescent lights in the ceiling had to be used for illumination, which led to less diffuse lighting and more glare from the shiny laminate floor. During the test run the drone wasn't moved in a square - it was such an early stage in the project that the experiment methodology hadn't been defined yet. Instead the drone was moved in a more Y-like shape. One final difference between the early test data and the final experiments was that the early test data was sampled at 5 Hz, which was lower than the 20 Hz of the offline samples from the experiment phase.

Figure 4.2 shows a navigation map after running the pipeline on the recorded early test data. To better show that the estimated path is continuous Figure 4.3 shows the same map as Figure 4.2, but with a line tracing the path of the drone via all the poses.

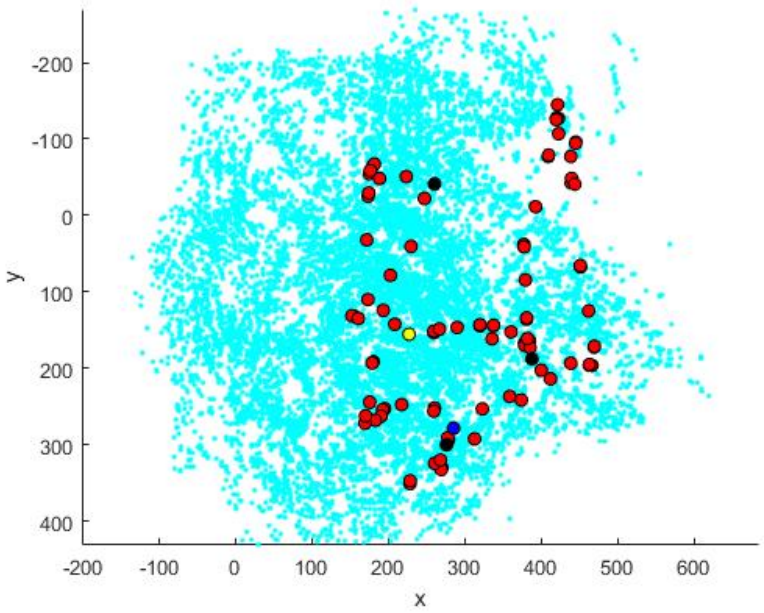


Figure 4.2: The estimated map and poses after running the pipeline on the recorded early test data. The drone was moved in a Y-like shape, instead of a square, and the estimated path was close to the true path. Cyan dots denotes SURF features, red circles denotes poses, black circles denotes poses from failed iterations[2], the yellow circle is the first pose, and the blue circle is the last pose. The relatively large gaps between the poses stems from the low sample rate of 5 Hz. The reason the poses seem shifted to the right w.r.t. the map is because the drone was accidentally held at an angle when it was carried.

[2]Recall that if an iteration failed the new pose was set to the last known pose, but no new features were added to the map.

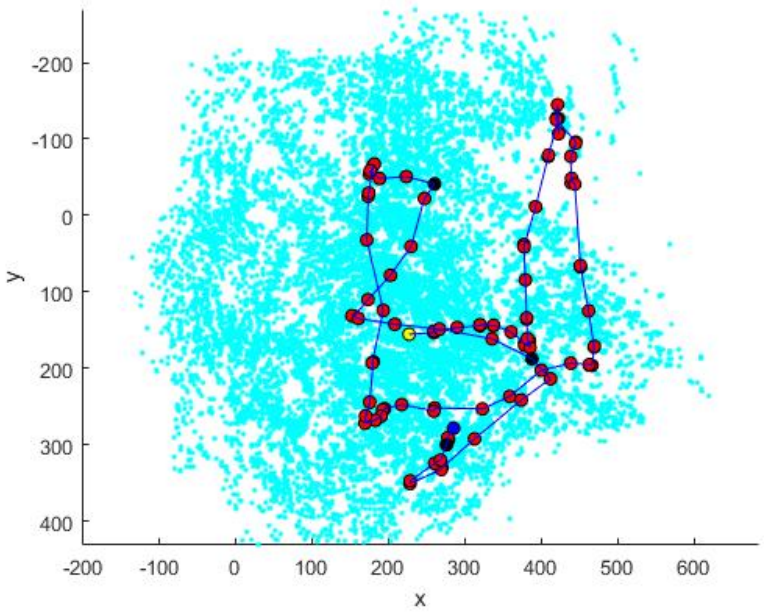


Figure 4.3: The same estimated map and poses as in Figure 4.3, but with a line drawn through all the poses to better show the estimated path of the drone. Notice that the path is continuous. The color scheme is the same as in Figure 4.2.

The robustness during the experiments was drastically worse than during the developmental tests on the early recorded data. Despite several attempts over several days, and testing with different drone velocities and altitudes, it was not possible to get a single run that didn't derail. Often the pipeline derailed after about 50 to 150 iterations[3], some time into the first maneuver command, i.e., along the first edge of the square. The most plausible reason for the pipeline derailing at those points was motion blur (caused by the drone either moving at top speed along the square's edges or rotating quickly as it maneuvered at the start or end of the movement command), but another possible factor was apparent lighting changes (caused by specular reflections and/or changes in exposure as the drone rotated during the maneuvers). It is important to note that the images from the bottom camera were never good to begin with, so these effects on top of the already low resolution, blurry image, and high compression was probably the straw that broke the camel's back and consistently made the pipeline derail. Figure 4.4 shows

[3]50 to 150 iterations translated to about 5 to 18 seconds, respectively.

a navigation map after a typical run during the experiment phase. That figure also demonstrates some other phenomena that have been mentioned previously, like the drone drifting and not being able to hover in place, and the drone shifting as it maneuvers to begin the forward strafe.

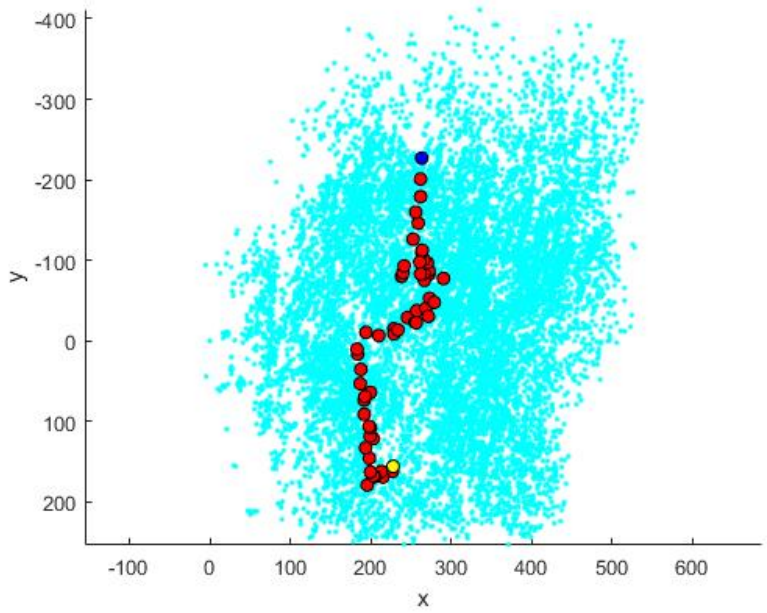


Figure 4.4: The estimated map and poses after a typical run during the experiment phase. The drone was commanded to fly in a square. After takeoff, the drone drifted forwards slowly before the command to strafe forward was issued, and then shortly after that the algorithm derailed. The bend halfway through the path was caused by the drone maneuvering as it began the strafe. The color scheme is the same as in Figure 4.2 and 4.3.

4.2 Computational performance

The performance (speed and efficiency) of the pipeline and the cost of the various subsystems could easily be measured using the MATLAB profiler, which gave data on execution times down to individual lines of code. From this data bottlenecks could be identified and the (inverse) performance scaling with map size could be investigated. The performance was measured on a Ubuntu system (running version 16.04 LTS) with 16 GB of RAM and a

Intel i5-4670K processor running at 4.5 GHz.

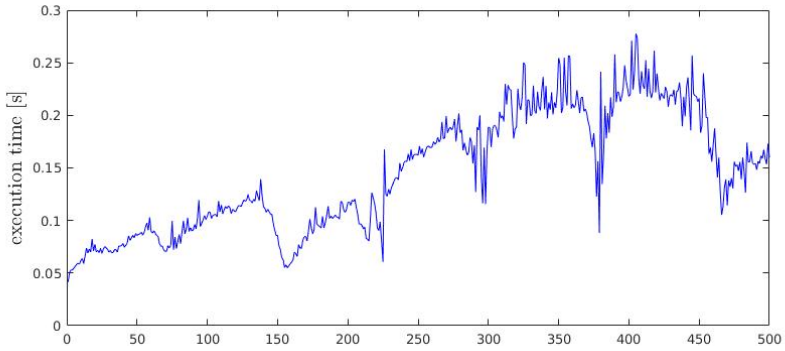
The performance was primarily evaluated on medium length campaigns with the drone repeatedly moving in a quasi[4]-square. The campaigns were repeated 5 times (using recorded campaign data) and results were computed as an average of the 5 runs. "Medium campaign length" here means 500 iterations, which equaled roughly 80 seconds and 2 completed "square laps".

Because of the robustness issues described in the previous section it was impossible to get a single medium length campaign that didn't derail. In order to study the performance of longer campaigns (50-150 iteration long campaigns, spanning 5 to 20 seconds, were not considered "long") the inlier criteria for registrations had to be lowered significantly so that the algorithm didn't derail, i.e., fail to estimate new poses. The resulting pose estimations were completely wrong, of course, but the long term performance of the pipeline could still be studied. These bogus estimations often placed the new poses in the densest part of the map, where the concentration of features was the highest and thus the probability of false matches the greatest. This in turn meant that the fast matching (geometric distance culling) was less effective than it would have been under more normal circumstances, due to the artificially high map density. This also made the bundle adjustment step more costly, as the number of poses related to the matched points was also artificially inflated from the high concentration of poses and features. Because of this, the performance tests can be considered "worst case" scenarios, and the pipeline performance would likely be better under less artificial circumstances.

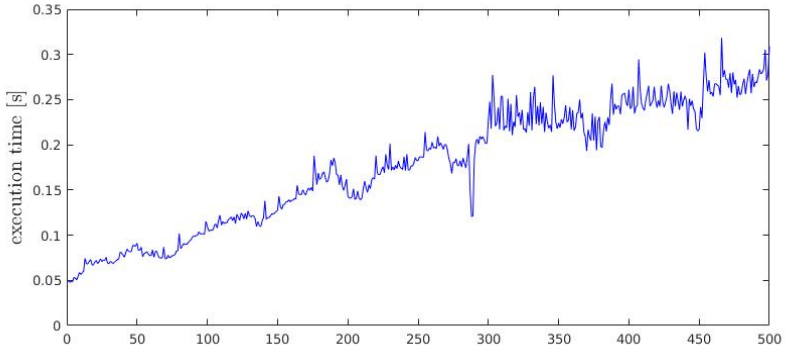
For a medium length campaign, as described above, the average execution time per iteration was 0.157 seconds, starting out at $\sim$0.05 seconds in the first iteration and ending at $\sim$0.20 seconds. The average final map size was $\sim$85000 features. The average amount of matches found between the last image and the map in each iteration was 118. It was hard to estimate how many of those matches were outliers, but given the robustness issues described in the previous section the ratio of outliers to inliers was likely large. Figure 4.5a shows a typical plot of the execution time as a function of the iteration number.

[4]See the point about maneuver precision at page 50.

The increase in execution time with iteration number mainly stemmed from more time consuming feature matching operations as the map grew, but the bundle adjustment step also became more expensive as the number of pose-feature relations in the map grew alongside the size of the map. The large fluctuations in execution time that can be seen in Figure 4.5a were iterations where the reduced map searched using the sped-up matching algorithm was especially small. If the last estimated location of the drone in the map was in a region with relatively few features then the smaller map searched with the sped-up matching algorithm would be very small and hence the feature matching step would be exceptionally fast.



(a) Representative execution times as a function of SLAM pipeline iteration # during a typical run.



(b) Representative execution times as a function of SLAM pipeline iteration # during a typical run with the *sped-up matching disabled*.

Figure 4.5: Representative execution times per iteration over 500 iteration long runs, with and without the sped-up matching step.

Feature matching

Of the total execution time almost 43.3% of the time was calls to MATLAB's built in feature matcher function, `matchFeatures`, which both the regular and the sped-up feature matching relied on. The reason it was so time consuming was the sheer size of the map that had to be searched in the later iterations. The sped-up matching algorithm helped, but it could only do so much due to how quickly the map grew even on the local scale when up to 300 features could be added to the map each iteration. For campaigns shorter than 500 iterations the feature matching would be less of a bottleneck due to the smaller map at lower iteration numbers.

Reading data, preprocessing and feature extraction

The next most expensive part of the pipeline was detecting and extracting SURF features, which stood for 10.1% of the total execution time. This cost (along with that of reading data from the drone and pre-processing the images, which stood for ∼4.0% of the processing time) could be considered overhead.

Bundle adjustment

The third most expensive part of the pipeline, which stood for 8.6% of the total execution time, was the bundle adjustment algorithm. Of the bundle adjustment algorithm's total execution time 48% of the time was spent on updating the map and poses after new optimal values had been found[5] and almost 24% of the time was spent on the initial assembly of the **J**, **H**, **e** and **v** arrays. The remaining 28% of the time was spent on calculating the LM-update, calculating the new errors, updating **J** and **H**, updating the variable vector, and on various overhead tasks.

The average amount of LM iterations per call to the bundle adjustment algorithm was 7.3 (including the first iteration), meaning that the cost of successive LM iterations was very low once the setup step in the first iteration was done. From the execution times reported via the MATLAB profiler a call

[5]The bulk of the map update computation time was spent on updating features that had only been detected once when their only related pose was updated, due to the cost associated with searching the map for all such features for each updated pose. Still, this time was very short with respect to the total computation time for the whole SLAM pipeline.

to the subroutine for the initial set-up of the sparse matrices was more than 6 times as expensive as all other calls in a single LM iteration put together.

Homography construction

The homography construction (including reading the odometry data from the drone) stood for less than 1.4% of the total execution time. Judging by this, updating the homography in each iteration using robot odometry readings (as described in Section 3.4.5) was not a significant performance drawback, compared to only computing it once, like what was done in [4] and [38].

Sped-up matching

The sped-up matching step had a small, but noticeable impact on the total execution time. This impact was measured by running the pipeline for 500 iterations over the same dataset (using the offline dummy drone) with and without the sped-up matching step engaged, and averaging the execution times over 5 runs. The average speedup was 21.8%, but given that feature matching was the single most expensive step of the pipeline this speedup was not negligible. The speedup in real-world scenarios would of course be variable depending on how the drone moved, which would in turn would how dense the map became. Figure 4.5b shows a representative plot of the iteration times of a single 500 iteration run with the sped-up matching disabled. Notice how the execution time there shows a stronger dependence on the iteration number and less extreme fluctuations than Figure 4.5a.

Early feature culling

The early SURF feature culling, where only the 300 strongest SURF features from the detector were passed on to the feature descriptor extractor, had a significant positive impact on the performance of the pipeline. In a typical image of the floor used in the experiments the SURF feature detector detected somewhere around 550 to 750 features, so culling this down to 300 or less made a significant difference. In order to study the exact impact this step had on the performance of the algorithm the pipeline was run for 200

iterations, while the drone flew in a square[6]. Using the dummy drone the experiment was repeated 5 times using the same data and the final map size and average execution time per iteration was averaged. Then the experiment was repeated (still using the same data via the dummy drone) but with the early feature culling disabled.

The result was that the average final map sizes were 13980 and 31120 features, for tests with and without early feature culling, respectively. Similarly, the average execution times per iteration was 0.0826 and 0.1987 seconds, respectively. In other words: the early feature culling led to a 56.0% decrease of the average final map size and a 58.5% decrease of the average execution time per iteration. On top of those performance increases the early feature culling also resulted in noticeably less noisy maps and poses.

[6]The reason for using 200 instead of 500 iterations here was that over the course of a 200 iteration long campaign the drone did slightly less than one complete lap. That meant that the image in each iteration was often of a new region, so map growth was more isolated than if views of previously visited regions were included, like in the length 500 campaigns.

Chapter 5

Discussion

5.1 Robustness

Here the driver for the ARDrone was launched via the ROS node AR.Drone 2.0 could be described as erratic at best and unreliable at worst. The exact sources of error were hard to pin down, especially since it was not possible to have the drone fly in a consistent manner.

The pipeline was most prone to derailing when the drone changed movement direction (and thus maneuvered and changed its tilt[1]) or slightly after a new movement command was issued and the drone was moving at or near top speed. The algorithm was generally very robust when the drone was just hovering - even if the drone rarely managed to stay stationary while hovering and often drifted around a bit. The reason for this stability was likely that the drone didn't tilt and moved slowly, causing the images to vary slowly from frame to frame and keeping the specular light in the images largely constant. It's also possible that the instability while maneuvering was due to problems with the odometry: bad odometry readings could lead to incorrect rectification homographies, which in turn could lead to failed pose estimations (this is discussed in more detail later in this section). This stability while hovering can be seen in Figure 4.4 where the drone hovers and drifts without the algorithm derailing but the algorithm derails shortly after the drone begins its strafe forward.

[1]The plastic laminate floor caused a non-negligible amount of specular reflections from the lights in the ceiling.

The most significant source of the lack of robustness was deemed to be the bottom camera of the AR.Drone 2.0. As was mentioned in Section 2.2, the camera was mainly intended to be used as a ground speed sensor (using optical flow). The camera was designed to provide the drone's on-board processor with low resolution[2], out of focus images at a high rate, and - as was learned during this project - using the camera for visual SLAM was pushing it a bit too far. The images from the camera were both out of focus, had low resolution and were significantly compressed. Motion blur was likely a factor too, but the images were already so blurry that it was hard to notice any difference when manually inspecting captured images.

It's possible that the camera was part of the reason for why the drone's maneuvers were inconsistent: if the drone was indeed using the bottom camera for ground speed measurements (which were in turn used for station keeping and maneuvering) then the odd maneuver behavior of the drone could very well stem from the camera. For example, the drone's inability to remain stationary while hovering could be due to incorrect ground speed measurements due to artifacts and changes in exposure of the bottom camera images. Indeed, a rolling effect in the image stream, moving from the top of the images to the bottom, was occasionally observed where the relative brightness of the image changed periodically even when the drone was perfectly still (landed on the floor). This could have led the drone into believing that it was moving in a way that it really wasn't.

Another possible source of the inconsistent behavior was deemed to be the odometry of the drone; even when the drone was stationary on the ground the odometry readings would fluctuate. This fluctuation was not very large and wasn't a problem for normal operation of the drone, but possibly had a negative impact on the SLAM pipeline when piled onto the previously mentioned shortcomings of the bottom camera. It's possible that jitter in the odometry occasionally resulted in incorrect tilt correction homographies which then in turn could result in incorrect or failed pose estimations. The drone's documentation [36] contained very little information about how the odometry readings were derived, but it seemed they relied on sensor fusion

[2]High resolution images would have been too demanding for the on-board processor to process at the required rate.

from the array of onboard sensors (gyroscope, accelerometer, magnetometer, ultrasound altimeter, pressure sensor, and ground-speed readings via bottom camera optical flow).

It's unclear whether sensor drift came into play during the experiments - the odometry was always zeroed before each test campaign by sending a so-called *flat-trim request* to the drone before taking off, telling it to zero its odometry frame assuming it was on a horizontal surface. The campaigns were usually short (less than 2 minutes) so sensor drift was unlikely to have impacted the results.

It was discovered that the bundle adjustment algorithm could occasionally backfire if a lot of false matches were made and registered as inliers. The subsequent optimization of the map and poses could then make the map worse, instead of better. This could be observed as an erratic repositioning of features and poses in the map by the bundle adjustment algorithm. In the worst case the algorithm would derail completely and poses and features would start to move together into one large mess in an attempt to reduce reprojection errors between all the false observations. This phenomenon was very rare and most often a knock-on effect of snowballing false registrations (see Section 4.1), so it was rarely the reason for the algorithm derailing, but instead occasionally made the derailings worse.

Finally, the elephant in the room when it comes to the robustness is why the early recorded test data gave robust results, but none of the experiments gave robust results. From the tests it seemed like motion blur and illumination were the two determining factors: the scenarios were different in the way the drone was maneuvered (carried by hand as opposed to flown via scripted maneuver commands) and the illumination of the scene (summer daylight as opposed to fluorescent ceiling lights). The first led to less motion blur and less jerky movements, and the second led to more consistent exposures and weaker specular reflections in the laminate floor. By comparing Figure 4.2 and Figure 4.4 it's clear that the overlap of the images from one frame to the next was lower in the early test data than in the experiment phase (cf. the distances between the poses in the two images and recall that the scale of the map was normalized using the altitude of the drone, so the map units are comparable). From this it is reasonable to assume that the amount of related features from one frame to the next was not a problem, but rather

the quality of those features.

During the experiment phase tests where made where the drone was carried instead of flown, in order to try to recreate the robust results from the early test. While the pipeline was more robust when the drone was carried around it didn't reach the same stability as when using the early test data. The pipeline still derailed eventually, but it managed to go on longer than when the drone was flown by scripts. when flying the pipeline usually derailed along the first edge of the square, and when the drone was carried the pipeline usually derailed along the second or third edge. Judging by this, either illumination played a key factor (since the summer daylight illumination could not be replicated during those tests), or some other, unknown factors played in - or the tests were just plain unlucky.

5.2 The algorithm

The algorithm was initially largely inspired by the work of Brange [4] and Rudbeck [38], but a lot of special considerations had to be made due to the different scenario and end-goal. Brange's work focused on offline map building using images taken of a planar surface with a constant tilt camera, and Rudbeck's work focused on using such a map to navigate using a drone equipped with a camera with constant tilt. The work in this thesis was focused on building such a navigation map from scratch and finding the drone's pose in it in real time, but with a consumer level drone with non-constant camera tilt. To this end the tilt correction homography had to be rebuilt each iteration and the map had to be built and maintained using only features, as opposed to Brange's map building algorithm which was built around matching and aligning images and thus used a bundle adjustment algorithm that was very close to the one employed by [13].

There were several sources of odometry readings made available through the ROS topics published by the drone and they all gave slightly different readings. The different sources were [26]: rotation about drone's (x, y, z)-axes via the `ardrone/navdata`-topic, (x, y, z)-rotation reported by the onboard IMU via the `ardrone/imu`-topic, (x, y, z)-rotation as part of odometry data via the `ardrone/odometry`-topic, and drone coordinate frame transformations (quaternion + 3 component translation) via the `tf`-topic. The documen-

tation did not clarify the differences between the readings from all those sources. The readings from the different sources were largely the same - the differences were on the scale of $< 1^\circ$ - so it was hard to determine which source was the most accurate. The choice of odometry source didn't impact the final rectification homography that much, but since the homography assembly already required reading transformations from the `tf`-topic in order to move from the bottom camera's coordinate system to the drone's local one it was opted to use the odometry readings from the `tf`-topic. Also, the whole point of the ROS `tf`-topic was to simplify moving between coordinate frames related to a robot, so using it to do just that made it feel like the right choice as the source of odometry readings.

While moving from a quaternion to Euler angles and then to a (transposed) rotation matrix (see Section 3.4.5) was more expensive than to going directly from Euler angles (from (x, y, z) drone orientation angles) to a rotation matrix neither approach had a significant performance cost. The transpose step could have been avoided if MATLAB's function for transforming Euler angles to a rotation matrix, `eul2rotm`, had supported the "XYZ" sequence, but the cost of transposing a 3×3 matrix was ultimately negligible compared to the total cost of each iteration in the SLAM pipeline.

5.3 Pipeline performance

Overall the pipeline performed very well - the initial goal was 5-20 Hz and the algorithm fell within that range for the campaign lengths tested during the experiments. The performance of the pipeline was very directly tied to the map size - the main bottleneck was the feature matching step where matches between the last image and the map were sought.

From the experiments (see Section 4.2) it was discovered that over the course of entire (500 iteration) runs 43.3% of the total execution time would be spent on calls to MATLAB's `matchFeatures` whereas reading images from the drone, preprocessing them and then detecting and extracting SURF features stood for less than 15% of the total execution time. The feature matching step was by far the single most demanding step of the pipeline so trying to optimize it is likely to provide a good return on investment. The performance could likely have been even better if better quality and higher resolution im-

ages were used: having to extract fewer (but higher quality and more salient) features from the images - and thus adding fewer features to the map in each iteration - would likely have improved performance more than it would have been decreased from having to detect and extract features from higher resolution images.

Another way to speed up the matching step would have been to use some sort of approximate matching method, like the one described in [27]. Hopefully, the performance benefit of such a method would outweigh the reduced amount and quality of matches returned due to the approximate nature of the method.

The bundle adjustment step was very effective under normal circumstances - its computation time only stood for 8.6% of the total computation time under the 500 iteration long campaigns done as part of the experiments. The most expensive part of the bundle adjustment step was (barring the post optimization map update, which required searching the map for single-detection features) the setup of the sparse matrices used to compute the Levenberg-Marquardt update. Once those matrices were set up they were very cheap to update after each iteration.

The main reason the sparse setup was so expensive was because no good vectorization was found for some of the sparse matrix initializations. All the sparse matrices used in the bundle adjustment algorithm were assembled using MATLAB's `sparse` function and three vector arguments[3] - which was a very effective function - but the filling in of the initial values in the triplet vectors couldn't be sufficiently vectorized in some cases. For example, no good vectorization[4] was found for filling in the vector containing all the original rotation matrix $\mathbf{R}_0$ (see Equation (3.25)) values (for all point-pose

[3]The syntax is `S = sparse(i,j,v)` where `i`, `j`, and `v` are arrays s.t. `S(i(k),j(k)) = v(k)`.

[4]One vectorization was found, but it was slower than the non-vectorized nested loop approach. That vectorization was to use MATLAB's `structfun` to get the rotation matrices for all poses, then extract the structure entries as a comma-separated list, which was then used as the argument of a call to `blkdiag`.

relations being optimized over, so 4 entries for each detection of each point)[5] due to the complex interactions between features in the map and the poses they had been detected from - instead a nested loop had to be used.

The poor vectorization of some of the components of the sparse matrix setup was generally caused by nested for-loops for unraveling the relationships between inlier features and the poses they had been detected from. Because of this the bundle adjustment step could become quite expensive when there were a lot of inliers that had been detected from a lot of poses. This could happen when the drone hovered and managed to stay relatively stationary, or when the SLAM algorithm was running while the drone sat on the ground. This could become a problem if the robustness issues were solved and the SLAM pipeline was used for longer missions in confined areas so that the same features would be detected very many times.

If the initial sparse matrix setup could be vectorized more it might be worth doing bundle adjustment across all poses and features, instead of only the inliers from each iteration and all their related poses. This large scale optimization could then be done in a separate thread at some interval - say, every 10th iteration. Another option would be to use some loop closure detection algorithm to flag when bundle adjustment should be performed.

[5]This vector was then used to set up a sparse matrix containing all the relevant $\mathbf{R}_0$-matrices along the diagonal, which could then be used to update the error vector $\mathbf{e}$ using just two matrix multiplications and two vector subtractions: i.e. $\mathbf{e} = \mathbf{x} - \mathbf{t} - \mathbf{A}^{spar}\mathbf{R}_0^{spar}\mathbf{X}$, see Section 3.4.10.

Chapter 6

Conclusions

The initial goal of this thesis project was to create a working and efficient SLAM algorithm for a quadcopter drone with a downwards facing camera. Under the course of the project it was discovered that the camera of the selected drone (an AR.Drone 2.0) was going to be a severe limitation for the SLAM algorithm, so then the goal was reformulated slightly so that the aim was to make the algorithm as good as possible using the given hardware. That goal was reached; the algorithm worked under the absolute right circumstances, as was evidenced by the experiments on the early captured data, but the algorithm generally didn't work in live tests where the optimal circumstances were hard to replicate. The algorithm was generally stable when the drone moved slowly, but due to the problems associated with giving the drone movement commands of less than 0.1 m/s (see Section 4) it was not possible to reliably keep the drone within that "reliability threshold" while exploring a scene. The main limiting factor was the drone's camera and the compression of the image stream; whether the pipeline worked or not seemed to be down to the quality of the images read from the drone as a result of the scene and the drone's maneuvering.

The iteration-wise update of the rectification homography worked well and didn't hamper the performance of the pipeline. The homography construction (including reading the odometry data from the drone) stood for less than 1.4% of the total execution time, so it was not a significant drawback compared to only having to compute it once. The accuracy of the on-board odometry was often sufficient, and the technique could likely be used to great effect for similar problem scenarios using better hardware.

The performance of the pipeline was good and fell within the range of $5 - 20$ Hz iterations for the campaigns tested as part of the experiments. That said, even with the sped-up feature matching step the pipeline would likely not have been suitable for longer campaigns due to how rapidly the map grew with each iteration. For longer campaigns it would have been necessary to slow down the growth of the map, but a different way to optimize the map w.r.t. to the measurements that required less metadata would possibly also be required.

The bundle adjustment step worked well and could be run each iteration without negatively impacting the performance of the pipeline. The cost of each iteration was very low - the total cost of each bundle adjustment call was dominated both by the cost of finding and updating the so-called single detection features in the map, and by the cost of the initial setup of the sparse matrices needed to compute the Levenberg-Marquardt update. *Single detection features* were relatively common due to the large amount of noisy features being added to the map that were never detected again and the initial sparse matrix setup was expensive due to the difficulty of vectorizing the complex feature and pose interactions embedded in the map metadata.

6.1 Future Work

The method for planar map construction using odometry sourced rectification homographies and point registrations was effective and could be developed further. The main limitations of the implementation were the hardware and the growth speed of the map. On the hardware side the drone's camera was the main hindrance, but the precision in the drone's maneuvers was also lacking. The growth speed of the map limited the algorithm to short campaigns of only a couple minutes in order to avoid the iteration frequency slowing down to less than 1 Hz.

Robustness

If the hardware was improved, e.g. by using a more modern quadcopter with a high quality camera, the advancements in both digital cameras and MEMS from the last 5 years would likely improve the robustness of the algorithm

greatly. A better camera would lead to better images, from which more inlier features (and less outliers) could be extracted, thus improving the pipeline where it needs it the most. Obviously, image compression would be a factor as well; heavy compression could render the high quality of the uncompressed image moot.

The pipeline could also be made more robust by storing images from failed iterations for some time and re-attempting to estimate the pose in those iterations at some later point in time. Currently, data from failed iterations is discarded and not used again, but it is possible that that data could be used later. For example, if some iterations failed when the drone entered a new area, the images from those iterations could possibly be registered to the map later if the drone moved back to a known area and the pipeline recovered. By going through those saved images in reverse order (newest first, oldest last) it could be possible to retrace the path of the drone and put information to use that would otherwise be lost.

Performance

If a better camera was used - as suggested in the previous section - then the resulting increase in inlier features/landmarks could also improve the growth rate of the map. A larger ratio of inlier features in each image reduces the need to extract many features to ensure that enough inliers are extracted to get a good pose estimation, thus slowing down the map growth as fewer features are added to the map each iteration.

To further increase performance and allow longer missions some better way of storing, searching and/or updating the map would be needed, even with the speed-up matching step. A different method of *storing* the map could be to use sub-maps, like in [16], where the full map is divided into several, partly overlapping maps and then only one sub-map is searched, based on the expected location of the drone. An alternative way of *searching* the map could be to use an approximate matching method, possibly the one described in [27], which employs tree searches and adaptive parameter selection to provide fast approximate high-dimensional nearest neighbor matching with configurable precision. The relatively expensive map *update* step at the end of the bundle adjustment could be optimized by adding some metadata to each pose. Specifically, a pointer to where in the map array the new de-

tections from each respective pose were saved could make it faster to find all the single detection features. Then only a small section of the map metadata array had to be searched, instead of the whole map metadata array.

The sped-up matching step could probably be sped up even further if sorted lists of all the coordinates of the features in the map (sorted by the x- and y-values, respectively) were used to find the relevant sub-map. In this case the L^∞-norm would have to be used, instead of the L^2-norm.

Bundle adjustment

The bundle adjustment is quite effective, barring some vectorization problems. If the vectorization could be improved, or if the sparse array setup was optimized in some other way it could be feasible to do global bundle adjustments over all poses and points. Such optimizations would suitable for running in a parallel thread and could be run more seldom than each iteration. Some possible timings for those calls would be either at a fixed frequency or whenever loop closures were detected, which would of course require some loop-closure detection step in the pipeline.

Map units

The units of the global coordinate system (and thus the map) was "scaled pixels". The pixels were scaled based on altitude measurements so that each pixel covered the same floor area. Using the measured altitude of the drone and the intrinsic camera parameters it would be relatively trivial to convert the map into real world map units, e.g. meters. The reason this wasn't done was mostly because of convenience; real world map units weren't required for the development of the pipeline and having more direct relationships between the three main coordinate systems (global, local and image) simplified troubleshooting.

Robot navigation

This project was exclusively focused on map building and localization. Robot navigation and path-finding was outside the scope except for the rudimentary navigation needed for the experiments. In any real world application, autonomous robot navigation would be an essential component of a system

employing a SLAM algorithm.

Rudbeck [38] developed a planar navigation algorithm which followed user defined waypoints in order to reach an end goal. That algorithm could be supplemented with a path-finding algorithm, such as A^* or Dijkstra's algorithm, to get a more complete navigation and mapping pipeline.

Lagrange multiplier rotation estimation

From private conversations with Magnus Oskarsson [33] another method for estimating the rotation during the RANSAC point registration (see Section 3.4.8) was suggested. Instead of the SVD-based method from [2] it was possible to solve for the optimal rotation matrix using a Lagrange Multiplier approach. This method was not implemented, even though it seemed superior on paper[1]. The reason for this was that the SVD approach worked sufficiently well and wasn't a bottleneck, so it wasn't prioritized in this project. Though, if this SLAM pipeline is developed further it might be worth implementing the Lagrange multiplier rotation estimation, just to make the pipeline even sleeker.

At the core of the Lagrange approach is a parameterization of the rotation matrix, $\mathbf{R}$, along with a constraint on the parameters:

$$\begin{aligned} \mathbf{R} &= \begin{bmatrix} a & -b \\ b & a \end{bmatrix} \\ \text{s.t. } & a^2 + b^2 = 1 \end{aligned} \tag{6.1}$$

The Lagrangian of the registration problem for two pairs of 2D points, $\{\mathbf{x}_1, \mathbf{x}_2\}$ and $\{\mathbf{y}_1, \mathbf{y}_2\}$, then becomes

$$L(a, b, t_1, t_2, \lambda) = \left(\sum_{i=1}^{2} \left(\mathbf{R}\mathbf{x}_i + \mathbf{t} - \mathbf{y}_i \right)^2 \right) + \lambda(a^2 + b^2 - 1) \tag{6.2}$$

Where $\mathbf{t} = (t_1, t_2)$ is a translation vector. The translation is removed from the equation in the same way as in the SVD approach, i.e. by subtracting the

[1] The SVD method was developed for least square fitting of large ($n \geqslant 2$) 3D point sets, so it was slightly overkill for the two-point 2D registration in this thesis. The Lagrange multiplier approach was more direct and had less overhead.

respective centres of gravity for the two point sets. From there the system is solved by expanding the square in the sum (exploiting that $\mathbf{R}^T\mathbf{R} = \mathbf{I}$), differentiating w.r.t a and b, then plugging in the values of $\mathbf{x}_i$ and $\mathbf{y}_i$ and solving for λ (exploiting that $a^2 + b^2 = 1$), and then finally solving for a and b.

Bibliography

[1] Aerix Drones. *Aerius - World's Smallest Quadcopter*. URL: `https://aerixdrones.com/products/aerius-the-new-worlds-smallest-quadcopter` (visited on 2017-12-23).

[2] K. S. Arun, T. S. Huang, and S. D. Blostein. "Least-Squares Fitting of Two 3-D Point Sets". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-9.5 (1987-09), pp. 698–700.

[3] H. Bay et al. "Speeded-Up Robust Features (SURF)". In: *Computer Vision and Image Understanding* 110.3 (2008). Similarity Matching in Computer Vision and Multimedia, pp. 346–359.

[4] E. Brange. *Efficient and robust map building from a downward looking camera with loop closing*. eng. Master's Thesis at Faculty of Engineering, Centre for Mathematical Sciences, Lund University. 2016.

[5] C. Chen and D. Schonfeld. "Pose estimation from multiple cameras based on Sylvester's equation". In: *Computer Vision and Image Understanding* 114.6 (2010). Special Issue on Multi-Camera and Multi-Modal Sensor Fusion, pp. 652–666.

[6] F. Chenavier and J. L. Crowley. "Position estimation for a mobile robot using vision and odometry". In: *Proceedings 1992 IEEE International Conference on Robotics and Automation*. 1992-05, 2588–2593 vol.3.

[7] E. Delgado and A. Barreiro. "Sonar-based robot navigation using nonlinear-robust Kalman filter". In: *2001 European Control Conference (ECC)*. 2001-09, pp. 1056–1061.

[8] J. Engel, J. Sturm, and D. Cremers. "Scale-aware navigation of a low-cost quadrocopter with a monocular camera". In: *Robotics and Autonomous Systems* 62.11 (2014). Special Issue on Visual Control of Mobile Robots, pp. 1646–1656.

[9] J. Engel and D. Cremers. "LSD-SLAM: Large-scale direct monocular SLAM". In: *In ECCV*. 2014.

[10] C. Fox et al. "Tactile SLAM with a biomimetic whiskered robot". In: *2012 IEEE International Conference on Robotics and Automation*. 2012-05, pp. 4925–4930.

[11] R. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Second Edition. Cambridge University Press, 2004.

[12] B. K. P. Horn, H. M. Hilden, and S. Negahdaripour. "Closed-form solution of absolute orientation using orthonormal matrices". In: *J. Opt. Soc. Am. A* 5.7 (1988-07), pp. 1127–1135.

[13] K. Konolige et al. "Efficient Sparse Pose Adjustment for 2D mapping". In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2010-10, pp. 22–29.

[14] J. Kosecka et al. "Qualitative image based localization in indoors environments". In: *2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings.* Vol. 2. 2003-06, II-3–II-8 vol.2.

[15] T. Lemaire et al. "Vision-Based SLAM: Stereo and Monocular Approaches". In: *International Journal of Computer Vision* 74.3 (2007-09), pp. 343–364.

[16] J. Leonard and P. Newman. "Consistent, Convergent, and Constant-time SLAM". In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence*. IJCAI'03. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, pp. 1143–1150.

[17] D. G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60.2 (2004-11), pp. 91–110.

[18] M. Magnabosco and T. P. Breckon. "Cross-spectral visual simultaneous localization and mapping (SLAM) with sensor handover". In: *Robotics and Autonomous Systems* 61.2 (2013), pp. 195–208.

[19] J. W. Marck et al. "Indoor radar SLAM A radar application for vision and GPS denied environments". In: *2013 European Microwave Conference*. 2013-10, pp. 1783–1786.

[20] MATLAB. *Computer Vision System Toolbox*. URL: `https://se.mathworks.com/products/computer-vision.html` (visited on 2017-10-05).

[21] MATLAB. *matchFeatures Documentation*. URL: `https://se.mathworks.com/help/vision/ref/matchfeatures.html` (visited on 2017-11-05).

[22] MATLAB. *Robotics System Toolbox*. URL: `https://www.mathworks.com/products/robotics.html` (visited on 2017-10-05).

[23] MATLAB. *What is Camera Calibration?* URL: `https://se.mathworks.com/help/vision/ug/camera-calibration.html` (visited on 2017-11-20).

[24] W. Meeussen. *ROS REP 105: Coordinate Frames for Mobile Platforms*. URL: https://www.ros.org/reps/rep-0105.html (visited on 2017-11-03).

[25] M. Monajjemi. *ardrone_autonomy Package Summary*. URL: http://wiki.ros.org/ardrone_autonomy (visited on 2017-10-05).

[26] M. Monajjemi. *Documentation for ardrone_autonomy - "Reading from AR-Drone"*. 2015-04-25. URL: https://ardrone-autonomy.readthedocas.io/en/latest/reading.html (visited on 2017-12-05).

[27] M. Muja and D. G. Lowe. "Fast approximate nearest neighbors with automatic algorithm configuration". In: *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009, pp. 331–340.

[28] P. Olson. *Dubai To Put Autonomous Taxi Drones In The Skies 'This Summer'*. Forbes. 2017-02-14. URL: https://www.forbes.com/sites/parmyolson/2017/02/14/dubai-autonomous-taxi-drones-ehang (visited on 2017-12-23).

[29] Open Source Robotics Foundation. *camera_calibration Package Summary*. URL: https://wiki.ros.org/camera_calibration (visited on 2017-11-03).

[30] Open Source Robotics Foundation. *geometry_msgs/TransformStamped - Documentation*. URL: http://docs.ros.org/api/geometry_msgs/html/msg/TransformStamped.html (visited on 2017-11-03).

[31] M. Oskarsson. *Lecture notes in FMA270, Computer Vision. Lecture 1: The Pinhole Camera Model*. 2017. URL: http://www.ctr.maths.lu.se/media/FMA270/2017/forelas1.pdf (visited on 2017-10-05).

[32] M. Oskarsson. *Lecture notes in FMA270, Computer Vision. Lecture 9: Local Optimization*. 2017. URL: http://www.ctr.maths.lu.se/media/FMA270/2017/forelas9.pdf (visited on 2017-11-05).

[33] M. Oskarsson. *Personal communication with Magnus Oskarsson*. Private Communication. 2017-11-06.

[34] Parrot SA. *Quadcopter AR Drone 2.0 Elite Edition*. URL: https://www.parrot.com/global/drones/parrot-ardrone-20-elite-edition (visited on 2017-12-23).

[35] L. M. Paz et al. “Large-Scale 6-DOF SLAM With Stereo-in-Hand”. In: *IEEE Transactions on Robotics* 24.5 (2008-10), pp. 946–957.

[36] S. Piskorski et al. *AR.Drone Developer Guide, SDK 2.0*. 2012-05-21. URL: `https://developer.parrot.com/docs/SDK2/` (visited on 2017-11-08).

[37] S. Riisgaard and M. R. Blas. *SLAM for Dummies: A Tutorial Approach to Simultaneous Localization and Mapping*. Tech. rep. 2005.

[38] F. Rudbeck. *Visual navigation and control of a mobile robot*. eng. Master's Thesis at Faculty of Engineering, Centre for Mathematical Sciences, Lund University. 2017.

[39] "Syahlevi". *AR.Drone 2.0 photgraph via Wikimedia Commons under the CC BY-SA 4.0 license; coordinate axes added by the author of this paper*. 2016-03-16. URL: `https://commons.wikimedia.org/wiki/File:81RNYV29HCL._SL1500_(1).jpg` (visited on 2017-11-28).

[40] M. Wadenbäck and A. Heyden. “Ego-Motion Recovery and Robust Tilt Estimation for Planar Motion Using Several Homographies”. In: *Proceedings of 9th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2014)*. SciTePress, 2014, pp. 635–639.

[41] M. Wadenbäck and A. Heyden. “Planar Motion and Hand-Eye Calibration Using Inter-Image Homographies from a Planar Scene”. In: *Proceedings of 8th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2013)*. SciTePress, 2013, pp. 164–168.

Appendices

A IMU covariance values

The three 3×3 covariance matrices for the IMU were stored in the 3 ROS parameters `cov/imu_la` (*linear acceleration*), `cov/imu_av` (*angular velocity*), and `cov/imu_or` (*orientation*). The following values were used for the covariance matrices:

$$\texttt{cov/imu\_la} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix}$$

$$\texttt{cov/imu\_av} = \begin{bmatrix} 1.0 & 0 & 0 \\ 0 & 1.0 & 0 \\ 0 & 0 & 1.0 \end{bmatrix}$$

$$\texttt{cov/imu\_or} = \begin{bmatrix} 1.0 & 0 & 0 \\ 0 & 1.0 & 0 \\ 0 & 0 & 100000.0 \end{bmatrix}$$